VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures

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

Ali Shafiee
University of Utah
Salt Lake City, Utah
shafiee@cs.utah.edu

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

Abstract

Intel's SGX offers state-of-the-art security features, including confidentiality, integrity, and authentication (CIA) when accessing sensitive pages in memory. Sensitive pages are placed in an Enclave Page Cache (EPC) within the physical memory before they can be accessed by the processor. To control the overheads imposed by CIA guarantees, the EPC operates with a limited capacity (currently 128 MB). Because of this limited EPC size, sensitive pages must be frequently swapped between EPC and non-EPC regions in memory. A page swap is expensive (about 40K cycles) because it requires an OS system call, page copying, updates to integrity trees and metadata, etc. Our analysis shows that the paging overhead can slow the system on average by 5×, and other studies have reported even higher slowdowns for memory-intensive workloads.

The paging overhead can be reduced by growing the size of the EPC to match the size of physical memory, while allowing the EPC to also accommodate non-sensitive pages. However, at least two important problems must be addressed to enable this growth in EPC: (i) the depth of the integrity tree and its cacheability must be improved to keep memory bandwidth overheads in check, (ii) the space overheads of integrity verification (tree and MACs) must be reduced. We achieve both goals by introducing a variable arity unified tree (VAULT) organization that is more compact and has lower depth. We further reduce the space overheads with techniques that combine MAC sharing and compression. With simulations, we show that the combination of our techniques can address most inefficiencies in SGX memory access and improve overall performance by 3.7×, relative to an SGX baseline, while incurring a memory capacity overhead of only 4.7%.

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3177155

CCS Concepts • **Security and privacy → Hardware-based security protocols**;

Keywords Security, memory integrity, Intel SGX, Compression

ACM Reference Format:
Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3177155

1 Introduction

A number of critical applications, e.g., electronic health records [22], are hosted in the cloud or in datacenters. Cloud systems must protect against a wide variety of attacks, including those launched by a compromised OS or by untrusted cloud operators with physical access to the hardware. Such attackers can snoop on signals emerging out of the processor, or can interfere with memory and processor inputs.

To protect against such attacks, a secure system must offer Confidentiality, Integrity, and Authentication (CIA) guarantees. Authentication is usually provided with hardware-enforced permission checks. Confidentiality is preserved by encrypting all signals that emerge from the processor. Integrity is the property that the memory system correctly returns the last-written block of data at any address. It is typically the most onerous guarantee because it requires the management and navigation of tree-based data structures on every memory access.

Intel has introduced Software Guard Extensions (SGX [12, 16]) that offer CIA guarantees for pages marked by an application as sensitive. SGX forms a secure hardware container, called an *Enclave*, to protect an application from several attacks, including those launched by an untrusted OS or by untrusted cloud operators. SGX partitions the physical memory into two regions: the *Enclave Page Cache (EPC)* that stores recently accessed sensitive pages, and a non-EPC region that stores non-sensitive pages as well as sensitive pages spilled out of the EPC.

The SGX memory controller is augmented with a *Memory Encryption Engine (MEE)* that performs permission checks, encryption/decryption, and integrity tree operations when

accessing any data block in the EPC. Therefore, EPC accesses are expensive. Sensitive pages in the non-EPC region have to first be moved into the EPC before they can be accessed. In the context of this work, *paging* refers to the process of moving pages between the EPC and non-EPC regions of physical memory. Non-sensitive pages in the non-EPC region are accessed without security overheads.

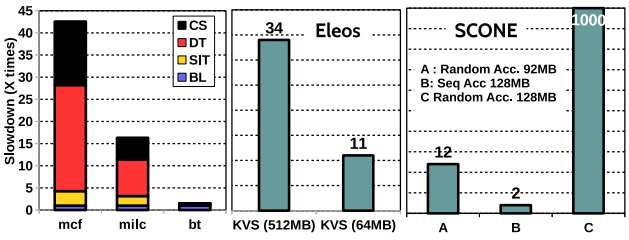

Figure 1. Left-side: slowdown for three different benchmarks with various numbers of page faults. The overhead is broken down in three portions, CS (Context Switch), DT (Data Transfer), and SIT (SGX Integrity Tree). The slowdown is against a non-secure baseline system (BL). Middle: the slowdown of SGX in a real system for a Key Value Store with two different working set sizes [33]. Right-side: slowdown for SGX in a real system for synthetic benchmarks, with random and sequential accesses, to different sizes of memory [2].

On modern hardware, the overheads imposed by SGX are very significant. A simulation-based analysis is shown in the left third of Figure 1, and has been corroborated on the right by measurements on real SGX hardware, reported by other papers [2] [33]. For a few memory-intensive applications, we see that marking all pages as sensitive can incur large overheads. Further, we break down this overhead into three components in the simulation-based analysis. The bottom blue component is the baseline non-secure execution time where none of the pages are marked sensitive. The top three components (in yellow, red, and black) represent overheads on every EPC hit and miss. An EPC miss is treated similar to a page fault, and requires an OS context switch (represented by the black sub-bar). The red portion of the bar represents the cost of moving a page between EPC and non-EPC, and corresponding updates of the integrity tree data structures. The yellow sub-bar represents the overhead experienced by every EPC hit – when accessing a block in a sensitive page in EPC, the integrity tree has to be navigated and updated.

In brief, there is a large gap between EPC hit and miss latencies – 200 cycles vs. 40 K cycles [2]. A recent software solution, Eleos [33], addresses the cost of OS context switches, but does not address the data transfer and integrity tree navigation costs.

Given these large paging overheads, an obvious follow-up question is: why not make the EPC larger to increase its hit rate? Intel SGX allocates only 128 MB for the EPC[1] . There may be a multitude of reasons for why the EPC is so small, some only known to industry engineers. We list some of the reasons here that are addressed by this work. (i) Integrity tree depth and size: the depth and size of the integrity tree grows with the size of the memory being protected. A large tree size and depth, in turn, lead to poor cacheability and higher bandwidth penalties when navigating the tree. (ii) Memory capacity overhead: the integrity tree and the message authentication codes (MAC) required by every EPC block can occupy a quarter of the memory being protected (32 MB out of the 128 MB). (iii) Workload demands: since EPC accesses are expensive, they are not appropriate for non-sensitive pages. Designating a large fraction of memory as EPC during design time may waste memory in applications that have few sensitive pages and under-utilize the EPC.

To enable a large EPC region, it is important to design integrity tree structures that impose lower bandwidth and capacity overheads, and can easily disable these overheads when non-sensitive blocks are part of the integrity tree. While we use SGX to motivate and frame the problem, our proposed integrity tree structures are generally applicable to any system that demands memory integrity.

We first introduce a *Variable Arity Unified encrypted-Leaf Tree (VAULT)* of counters for integrity verification that efficiently manages the trade-off between tree depth and counter overflow. While Intel SGX has a tree with arity 8, VAULT is designed to have a variable arity of 16 to 64. By flattening the tree, and by making it more compact, the cacheability and bandwidth overheads on every read and write are greatly improved.

Second, we propose a technique (SMC) that uses compression to pack a data block and its MAC into a single cache line, thus reducing bandwidth overheads. Further, we reduce storage overheads by sharing a MAC among multiple data blocks. While this approach has the potential to increase memory bandwidth demands, we show that the compression-based technique can eliminate or reduce the bandwidth penalty in most cases. Thus, SMC can reduce both bandwidth and memory capacity. Finally, we allocate MACs on-demand just for sensitive pages to further reduce MAC capacity overheads.

With these techniques in place, the EPC can be expanded to cover the entire physical memory with tolerable bandwidth and capacity overheads. With help from the TLB, non-sensitive pages can disable subsets of their CIA operations and not be penalized. Even if a large fraction of pages are non-sensitive, the integrity tree overheads for sensitive

[1]In SGX, the 128MB is called PRM (*Processor Reserved Memory*) in which 96MB is for data (called EPC) and the rest is used for metadata. For simplicity, we use EPC to describe both of them in this work.

pages are tolerable. Most importantly, when the sensitive working set scales up, there is no penalty from paging.

Our results show that baseline SGX with paging, and Eleos incur an average slowdown of 5.55× and 2.43× respectively, relative to a non-secure baseline. The capacity overhead in the baseline is under 1%. If SGX is naively extended with an EPC as large as physical memory, it incurs a slowdown of 1.71× (from integrity tree navigation) and a capacity overhead of 25%. With VAULT, SMC, and on-demand MAC allocation in place, and an EPC as large as physical memory, we experience a slowdown of 1.5× and a capacity overhead of less than 4.7%. Non-sensitive pages can be accessed without any bandwidth overheads, and sensitive pages are allowed to have a working set as large as physical memory.

2 Background

2.1 Threat Model

Physical Attacks in the Cloud. In cloud computing environments, applications are executed on remote servers. The hardware platform is therefore managed by a potentially untrusted cloud operator. This renders the system vulnerable to physical attacks, where the attacker can replace hardware modules, e.g., DIMMs, with specialized modules that can snoop on data, modify data, or engage in denial of service. With physical access, an operator can also install a malicious OS that can tamper with application data by taking ownership of the application's pages.

Software Attack Model. We assume that attackers have full control over different levels of the software stack including OS and any other programs. The OS or any malicious applications can attempt to compromise data confidentiality and integrity. This work does not address any side channel attacks, and memory safety bugs (e.g., buffer overflow). Denial-of-service attacks are also out of the scope of this study.

Physical Attack in the Memory System. We will assume that the host processor is a secure entity, i.e., it employs best practices to protect its internal data and does not leak information through side channels. But such a secure processor must eventually store results in main memory or disk. We will focus on the more common memory transactions in this work. Memory transactions are performed on DDR memory channels that are visible on the board and that can be snooped with logic analyzers. Alternatively, an attacker can design a custom DIMM with a buffer chip that acts as a liaison for all exchanged signals, and therefore has full access to all exchanged data. In short, since the attacker can control the hardware and OS, they can access and control all information going in/out of the secure processor. This is true regardless of whether the memory is implemented with DDR standards or emerging protocols like that in Micron's Hybrid Memory Cube [21].

Guaranteeing Confidentiality with Encryption. To prevent attackers from snooping on externally visible data, and guarantee *confidentiality*, a secure processor can encrypt all data packets emerging from the processor. Memory devices store data blocks in their encrypted form and simply return the last-written copy when the block is requested again.

MACs to Thwart Some Integrity Violations. While the attacker cannot violate confidentiality, they can violate the property of *integrity*, which guarantees that the processor receives exactly the same contents that were last written into a memory block. When the processor requests data from an address, the attacker can return a randomly created block of data. This is easy to detect. Every block of plaintext can be associated with a *Message Authentication Code (MAC)*, which is typically a 64-bit field (in the case of SGX) produced by applying a hashing function on the plaintext. When the encrypted data and MAC are fetched from memory, they are first decrypted, the MAC for the plaintext is re-computed, and this MAC is matched against the MAC received from memory. If the attacker has created a random block of data, with a very high probability, the processor can detect that the block is corrupted. The encryption/decryption function can also incorporate the block address so that the attacker cannot perform a *splicing* or *relocation attack*, where they return a valid block/MAC combination resident at a different memory location.

Another Integrity Violation – The Replay Attack. In spite of using the MAC, the system is still vulnerable to a *replay attack*. In a replay attack, the attacker returns a block/MAC that was previously written to a given memory location, but is not the last write. Such a block/MAC, after decryption, will pass the MAC confirmation. This is the type of attack that integrity trees, including that of SGX, are attempting to thwart. We will first briefly review Merkle and Bonsai Merkle Trees that have long been used for replay attack defenses. We will then describe the mechanisms used in SGX, the state-of-the-art industry baseline.

2.2 Merkle Trees

In a Merkle Tree (MT), the MACs of all the data blocks represent the leaf nodes. Each non-leaf node stores a hash of its child nodes. The root of the tree is maintained on the processor. Assuming a 64-bit MAC or hash similar to that in SGX, eight MACs/hashes can fit in a single 64 B cache line. As a result, the tree is organized with an arity of eight. Thus, a single cache line fetch can retrieve the eight children of a node. On every data block read by the processor, all ancestors of the block's MAC have to be fetched from memory; the MACs/hashes are verified on the processor; if the attacker attempts a replay attack, at least one of these will yield a mismatch. Because of the relatively low arity, the MT has a high depth, e.g., a 16 GB memory requires a

10-level MT. In other words, every memory access in a non-secure baseline translates to 11 memory accesses when using an MT (MACs and hashes can be cached, and this will be considered throughout).

All these blocks can be potentially fetched in parallel and the processor can speculatively proceed with the data block while the verification can happen in the background [25]. But several modern workloads are already memory-intensive and most enterprise systems operate their memory channels near saturation. Therefore, while the latency of a single Merkle Tree fetch can be hidden, the bandwidth overhead will have repercussions. If the memory channel in a non-secure baseline is already highly utilized, a 11× bandwidth overhead will manifest as a 11× application slowdown. Therefore, it is critical to reduce the bandwidth overhead. Note that a write to a data block requires us to read all its ancestors in the MT, followed by a write to all those ancestors, i.e., the bandwidth overhead of a write is nearly twice that of a read. Some of the above overhead can be alleviated with caching. It is reasonable to expect the processor's LLC to accommodate between six to eight levels of the top of the Merkle Tree.

2.3 Bonsai Merkle Trees

Tamper-Proof Counters to Prevent Replay. To alleviate the high overhead of Merkle Trees, Rogers et al. [35] introduced the concept of Bonsai Merkle Trees. It borrows many of the same principles as a Merkle Tree, but adds the following new feature. Just as we used the block address in the encryption/decryption function to prevent the attacker from returning valid data/MAC at a different address, we can also use a version number in the encryption/decryption function to prevent a replay attack. Thus, for every block, we need a counter (or version number) to keep track of how many times this block has been written, and this counter is required during the encryption/decryption process. Millions of counters cannot be accommodated on the processor chip, so these counters will eventually have to be stored to and retrieved from memory. Therefore, during a read, we must fetch the data block, its MAC, and its counter; the counter is used for decryption; the MAC is computed to confirm that the block is valid. But an attacker can perform a replay attack by returning an old block, old MAC, and old counter. Thwarting any of these three returns is enough to preserve data integrity. To prevent the attacker from returning an old counter, we can maintain a Merkle Tree on the counters, i.e., the leaves of the Merkle Tree are 8-bit counters for all data blocks, not the 64-bit MACs for all data blocks. This simple change results in a *Bonsai Merkle Tree (BMT)* that has 1 fewer level than a Merkle Tree. The memory storage overhead of the BMT is small; in fact, the metadata storage is dominated by the 64-bit MAC that is maintained for every 512-bit block, i.e., a storage overhead of 12.5%.

Managing Shared Counters for High Security and Low Overhead. One problem with this approach is that when a counter reaches its maximum value and cycles back to zero, it is vulnerable to a replay attack, i.e., the attacker can return an old block that can be correctly decoded with the current counter value. Therefore, counter values should never be recycled. To enable this, the leaf nodes of the BMT are reorganized. Instead of placing 64 8-bit counters in a cache line, the BMT places 64 7-bit (local) counters in a cache line. There is also room for a shared 64-bit global counter that serves as a prefix for all local counters in that cache line. That is, every data block is now represented by a 71-bit counter. When any local counter cycles back to zero, the shared counter is incremented, thus always yielding unique 71-bit counters for a given data block during the reasonable lifetime of a system. When the global counter is incremented, since it is shared, all 64 blocks represented by that node have to be re-encrypted with their new counter value and written back. We also take this opportunity to zero out all 64 local counters in that node. This approach addresses the replay vulnerability, but introduces an overhead (of 64 reads and 64 writes) every time a local counter cycles back (overflows). As we show later, this overflow overhead is relatively small in the BMT, but can be significant for other tree organizations.

2.4 Intel SGX Baseline [12]

SGX Overview. SGX partitions the main memory into two parts: EPC (Enclave Page Cache) and non-EPC. The enclave created for an application can include both sensitive and non-sensitive pages. When the application requests the OS for a sensitive page, it is mapped in the EPC. To protect the EPC, the CPU is responsible for enclave authentication as well as performing TLB checks to prevent the OS from TLB-base attacks. In addition, the Memory Encryption Engine (MEE) encrypts/decrypts data blocks (confidentiality) and ensures data freshness using an integrity tree (integrity and message authentication). The EPC has a small 128 MB capacity, of which, 32 MB is used to store the MAC for each block, as well as the integrity tree structure (which we will describe shortly), and some other metadata for each EPC page.

When a sensitive page is evicted out of the EPC, it is stored in the non-EPC region. CIA guarantees must be provided for sensitive pages in the non-EPC region as well. Upon evicting from the EPC, MEE decrypts the page and hands it to the CPU. The CPU then assigns a counter, encrypts the page using the combination of the counter and the enclave's key, and calculates a 128-bit MAC for the entire page. The encrypted page is inserted into a non-EPC integrity tree (called the eviction tree) to guarantee that any tampering of these sensitive pages can be detected. To reduce its overhead, the eviction tree works at the page granularity. Note that individual blocks of a sensitive page

cannot be accessed unless it is moved back to the EPC. When an application wants to access a sensitive page, it is moved into the EPC after the CPU has authenticated the request and verified it using the eviction tree.

The sensitive pages can take advantage of the eviction tree even when they are moved to the swap space. The non-EPC region also stores non-sensitive pages without CIA guarantees.

Protecting from TLB Manipulation. In SGX, page tables and extended page tables are fully under the control of the OS or the hypervisor. As a result, a malicious OS can allocate or redirect an unexpected physical page to a virtual page, which leads to unintended inputs or a change in the program's control flow (active memory mapping attack). To protect from such attacks, SGX maintains an entry of metadata for each sensitive page in an array called Enclave Page Cache Map (EPCM). Every EPCM entry has ADDRESS and ENCLAVESECS fields; the former contains the virtual address assigned to the corresponding EPC page while the latter keeps track of the sensitive page's owner. SGX uses these fields when handling a TLB miss, to avoid any TLB manipulations for senstive pages.

After the TLB translates the virtual address to a physical address, the secure CPU uses the physical address to find the appropriate EPCM entry. It then authenticates the requesting enclave using the ENCLAVESECS field and matches the corresponding virtual address with the ADDRESS field.

It is worth noting that SGX limits the virtual address space assigned to senstive pages to a range known as ELRANGE (Enclave Linear Address Range). SGX treats the pages outside this range as non-sensitve and disallows allocating them to any EPC pages.

Paging Overheads [33]. In SGX v2, sensitive pages can be allocated to enclaves dynamically. When an enclave encounters a page fault, i.e., the requested page does not exist in the EPC, the enclave is forced to exit, a context-switch to OS occurs, the requested page is moved to the EPC, and control returns back to the enclave. Unfortunately, this process imposes a significant overhead on performance due to two main reasons: OS-related and data transfer overheads. OS-related overhead includes exiting and re-entering the enclave (through EEXIT and EENTER instructions), flushing the TLB, context switching, and handling the page fault. Data transfer overhead is due to data transition and integrity checks between the EPC and non-EPC parts. The total paging overhead is around 40K CPU cycles.

SGX Integrity Tree (SIT). We now discuss the integrity tree algorithm used by SGX for data blocks in its EPC. Similar to the BMT, every block in the EPC region is associated with a counter. But instead of building a tree of hashes on top of these counters, SGX designs a new tree structure, shown in Figure 2, that we dub *SIT*. Every 512-bit node of

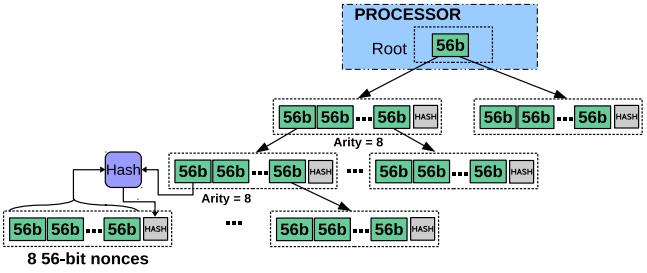

Figure 2. SGX integrity tree (SIT).

the tree is composed of 8 56-bit counters and a 64-bit hash[2]. The hash in a node is a function of the 8 56-bit counters in that node, as well as one 56-bit counter in the parent node (using the Carter-Wegman algorithm [40]). This sets up the dependency between child and parent, which must be verified from the leaf node all the way up to the root. The SIT has an arity of 8 throughout; recall that the BMT has an arity of 64 at the lowest level and an arity of 8 for all higher levels.

Read/Write Example. On a read, we fetch the data block, its MAC, and its corresponding 56-bit counter. We then fetch the ancestors of that counter from SIT (until a cache hit). All of these fetches can happen in parallel, leveraging all the available parallelism in the memory system. For each level i of the SIT, the processor confirms that the 8 counters in level i and the corresponding counter in the parent level $i-1$ produce a hash that matches the hash in level i. If the attacker attempts some kind of replay, at least one of the hashes or MAC will disagree with a very high probability.

When a block is written, the counter for that block and all its ancestor counters (until a cache hit) must be incremented. The corresponding hashes will also have to be updated. This requires a read of the counter node and all its ancestors (until a cache hit), followed by writes to the same nodes.

3 Proposed Techniques

3.1 Unifying the EPC and non-EPC Regions

To eliminate paging overheads, we eliminate the demarcated EPC and non-EPC regions, and simply define a single unified physical memory. Within this unified physical memory, some pages may be marked sensitive, while others may be marked non-sensitive. This sub-section discusses how the hardware determines if a page is sensitive or not, how to authenticate the enclave, and protect from memory mapping attacks. The next sub-section discusses the integrity check operations in case the page is sensitive.

The basic idea is to allocate one EPCM entry for every physical page in the main memory. EPCM is updated by the secure hardware to prevent the OS from tampering with

[2]While SGX uses a 56-bit hash, without loss of generality, we model SGX with a 64-bit hash.

metadata. As described in Section 2.4, every EPCM entry includes information regarding the enclave owning the page, as well as the virtual address bound to the physical address. We also augment the entry with a field, named SENSITIVE, to indicate whether the page is sensitive or not. Note that similar to SGX, EPCM is stored in sensitive pages.

When accessing a page, a TLB look-up translates the virtual address to a physical address. The secure CPU uses the physical address to fetch the corresponding EPCM entry. For this entry, if the SENSITIVE field is not set, then the CPU performs a regular memory access, similar to a non-secure memory system. Otherwise, similar to SGX, the CPU matches the translated virtual address against the entry's ADDRESS field. The final sanity check is to authenticate the enclave, i.e., the CPU compares the ownership information of the page (field ENCLAVESEC in the EPCM entry) with the ID of the requesting enclave. In the case of any mismatches, a general protection fault happens.

Similar to TLB entries, the EPCM entries can be cached in a hardware structure that only the secure CPU can access. Therefore, for a TLB hit, the corresponding EPCM entry is also available on the chip. However, a TLB miss takes longer, compared to a non-secure system, as it requires fetching an EPCM entry from a sensitive page. The table with EPCM entries represents a negligible capacity overhead of much less than 0.1% in physical memory because each entry only requires 16 bytes.

When accessing a non-sensitive page, the typical encryption and integrity checks can be elided and non-sensitive page accesses are as fast as those in a non-secure baseline. This concept can be further generalized – multiple bits in the SENSITIVE field of the EPCM entries can define multiple security levels, some that enforce only authentication and confidentiality, others that enforce CIA guarantees, etc.

As mentioned in Section 2.4, the OS might transfer a page to the swap space. Therefore, for trusted pages, CIA should also be guaranteed on the swap space. In SGX, the eviction tree provides CIA for both, the non-EPC part of the main memory and the swap space. In our approach, the entire main memory is protected by a unified tree (Section 3.2), while the eviction tree is shifted to cover merely the swap space.

Next, we introduce more efficient approaches to provide integrity for the entire physical memory. If the same integrity tree (SIT) used for the baseline 128 MB EPC is now used for the entire physical memory, there are two major overheads: (i) The depth and size of the tree would be much greater, thus incurring a significant bandwidth penalty for every sensitive block access. (ii) The metadata overheads would grow from 32 MB to several giga-bytes.

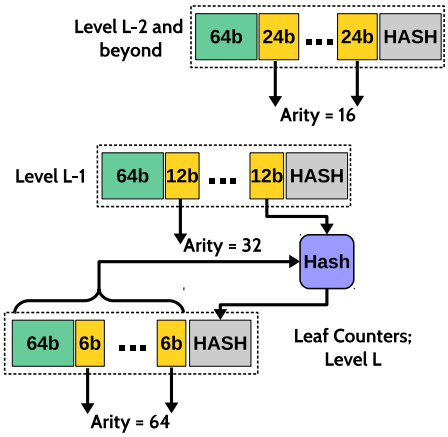

Figure 3. Variable Arity Unified Tree (VAUT).

3.2 Variable Arity Unified encrypted-Leaf Tree (VAULT)

We first describe a new integrity tree organization, VAUT, that improves tree depth, tree size, tree cacheability, and hence the bandwidth overhead. The proposed integrity tree is unified because it includes all blocks, sensitive or not, in physical memory. Similar to the SIT organization, a 64-bit hash in a node is computed based on the other 448 bits in that node and a sufficiently large counter in the parent (see Figure 3). This hash establishes a hard-to-fool linkage between parent and child in the tree.

The key to flattening the integrity tree is an increase in its arity. The BMT has an arity of 8 by placing 8 64-bit hashes in a cache line. The SIT achieves an arity of 8 by placing 8 56-bit counters in a cache line. We adopt the same linkage organization as SIT, but place even more counters in a cache line.

We first construct a strawman where every node of the tree maintains a 64-bit hash and 64 7-bit counters. By using many small counters, we achieve a tree with arity 64 and a depth of only 5 for a 64 GB memory (with the top two levels potentially being cached on the processor chip). While this makes the tree access dramatically more efficient, it causes the counters to cycle back to zero after 128 memory accesses, making the system vulnerable to replay attacks. Therefore, the tree must be designed to balance arity/depth and counter overflows.

Figure 3 shows our proposed VAUT organization. Similar to the BMT, we use the concept of shared global counters and local counters in every node. At the lowest level of the tree, a leaf node maintains a 64-bit hash, a 64-bit shared global counter prefix, and 64 6-bit local counters. In other words, we are maintaining 64 70-bit counters in a node, but all of these counters share the same 64 most significant bits.

When any of the 64 local counters cycles back to zero, we increment the global counter and reset all 64 local counters in that node to zero. Such a reset requires us to re-encrypt all

the data blocks corresponding to that node, thus incurring an overhead of 64 reads and 64 writes. In the BMT, where the leaf node maintains 7-bit local counters, this reset overhead is incurred when a local counter value reaches 128. In the proposed organization, the local counters in the leaves reset when they reach 64, i.e., the reset overhead may be two times as high and noticeable.

If we preserved the same node structure at all levels of the tree, we also have to worry about reset overheads at other levels of the tree. The local counter in a node is incremented when any data block in its subtree is updated. This means that the higher levels of the tree (if uncached) increment their counters far more frequently than lower levels of the tree. If all nodes in the tree follow the same organization as the leaf node, the 6-bit counters in higher levels of the tree will cycle back to zero very frequently, and incur the high reset overhead (64 reads and 64 writes) on each reset[3]. Note that the BMT did not have to deal with this problem; it used global and local counters only for leaf nodes; the non-leaf nodes were composed of hashes, not counters.

To keep the reset overhead in check, we must allocate more bits for each of the local counters in a node, as we move to higher levels of the tree. This is illustrated in Figure 3, where the parent of a leaf node has a 64-bit hash, a 64-bit global counter prefix, and 32 12-bit counters. The grandparent of the leaf node and its ancestors have a 64-bit hash, a 64-bit global counter prefix, and 16 24-bit counters. Thus, the higher-level nodes that are much more vulnerable to reset overheads are provided with significantly larger counters, yielding a tree with arity 64 at the lowest level, arity 32 at the level above, and arity 16 for higher levels of the tree. The top levels of the tree are likely to see even more counter updates, but they are also much more likely to be cached – note that counter increments are not required as soon as we encounter a cached node of the tree. Therefore, it is not necessary to allocate more than 24 bits per local counter for levels higher than the grandparent of the leaf. This variable arity tree has a depth of 7 for a 64 GB memory; note that SGX's tree depth is 10 and BMT's tree depth is 9 for the same memory capacity.

Another side effect of VAUT is that the use of more space-efficient counters results in a smaller tree, relative to SIT structures (1.6% vs. 12.5% of the total memory capacity), that in turn leads to better hit rates in the processor's cache. The higher cache hit rate for VAUT nodes can reduce memory bandwidth and reset overheads.

VAUT with encrypted Leaves (VAULT): The biggest drawback of the VAUT technique is that it only allocates 6 bits per local counter in leaf nodes, causing a noticeable number of resets. Each reset overhead is also highest at the leaf level because it involves 64 reads and 64 writes (a reset in the parent of the leaf involves 32 reads and 32 writes). Further, as we show in the next sub-section, some of the leaf node bits may be required for other metadata. If each local counter were to receive only 5 bits, the reset overhead would essentially double. Therefore, to manage reset overheads in the leaf node, it is important to somehow grow the size of each local counter.

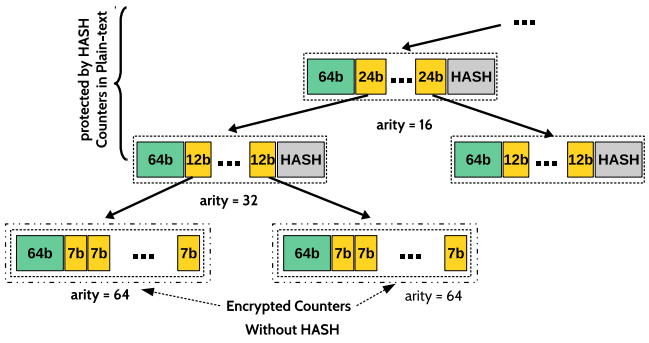

Figure 4. VAUT with encrypted Leaves (VAULT)

For the leaf nodes in VAULT, we eliminate the 64-bit hash field. Recall that in VAUT, the counters in the leaf were combined with a 76-bit counter in the parent to produce the hash in the leaf. If we eliminate the hash, we need an alternative method to establish a linkage between leaf and parent. This linkage is established by using the 76-bit counter in the parent as a key to encrypt the leaf block. If an attacker tries to fabricate either the leaf or the parent, the decryption of the leaf block would likely yield an incorrect 71-bit leaf counter (we analyze this further in Section 3.5), which in turn would likely yield an incorrect data block that fails the MAC confirmation – note that every data block in BMT, MEE, and VAULT is still associated with its own separate MAC. By eliminating the hash in the leaf node, every local counter can be 7 bits instead of 6 bits, which reduces reset overheads by roughly a factor of 2×. This organization is shown in Figure 4, and is referred to as a VAUT with encrypted Leaves (VAULT).

However, there is one drawback to this approach. The decryption of the leaf block is now on the critical path of the MAC confirmation, adding 40-80 cycles [19] to the MAC confirmation latency. This is also why the encryption-based approach should only be used where it is most required – at the leaf nodes that suffer from high reset overheads. It should not be employed at higher levels of the VAULT.

3.3 Shared MAC with Compression (SMC)

In the VAULT technique, the tree has a depth of 7 for a 64 GB memory, with the top levels of the tree potentially cached on the processor chip. A memory access may therefore require fetching the bottom three levels of the tree, the data block itself, and its MAC. Since we have reduced

[3]A reset in a non-leaf node requires an update of the hash in all its child nodes. A reset in a leaf node requires a re-encryption of all corresponding data blocks.

the tree access overheads, the MAC overhead is now noticeable and worth reducing. We reduce this overhead with a compression-based approach that meshes well with the VAULT design.

Before a data block is encrypted, we first compress the block. If the 512-bit data block can be compressed to 448 bits or less, the unused tail of the block can be used to accommodate the block's 64-bit MAC. Therefore, instead of separately fetching the data block and its MAC, a single block fetch can yield the data and its MAC. This can reduce the bandwidth requirements when dealing with compressible blocks. However, we need one additional metadata bit per block to track if a block has been stored in compressed or uncompressed form. This bit can be stored along with the block's local counter in the leaf node of VAULT. This reduces the local counter size from 7 to 6, introducing a trade-off between reset overhead and memory bandwidth. The compression-based approach further increases the critical path for MAC verification (since the compression bit is required before fetching the MAC). As our results show, this is a worthwhile trade-off because memory-intensive applications are more sensitive to bandwidth increase than to latency increase. Also, the processor can speculate and move ahead with the data block while the verification is performed in the background [25].

But memory capacity is also an important metric that must be improved. As described earlier, the MACs introduce non-trivial storage overheads of 12.5% in BMT and SGX's EPC. To reduce this capacity overhead, we share a MAC among multiple blocks. If a MAC is shared among 8 or 4 blocks (referred to as a *group*), the MAC storage overhead can drop from 12.5% to a more palatable 1.6% or 3.1%, respectively. However, MAC sharing can lead to an increase in bandwidth requirement, especially when spatial locality is limited. To verify a MAC, all the data blocks in the group would be required.

To address this storage vs. bandwidth trade-off, we again leverage our compression-based scheme. As shown in the example in Figure 5, a group of 4 data blocks, $D0 - D3$, share a MAC $M0$. But if $D0$ is compressible, it maintains a private MAC $m0$ that is co-located with data in a single block. Similarly, $D2$ is also compressible in this example and maintains a private MAC $m2$. The shared MAC $M0$ therefore only involves blocks $D1$ and $D3$. Thus, by combining a Shared MAC and Compression (a technique we refer to as SMC), we lower storage overheads and reduce the bandwidth requirements. When accessing compressed block $D0$, a single block can provide the data and the MAC. When accessing uncompressed block $D1$, we must fetch blocks $M0$, $D1$, and $D3$ – by examining the compressibility bits for the group, we can avoid fetching all the blocks in the group. Note that we have used compression here to reduce bandwidth demand; compression has not been used to reduce overall memory capacity requirements. As seen in Figure 5, compressible

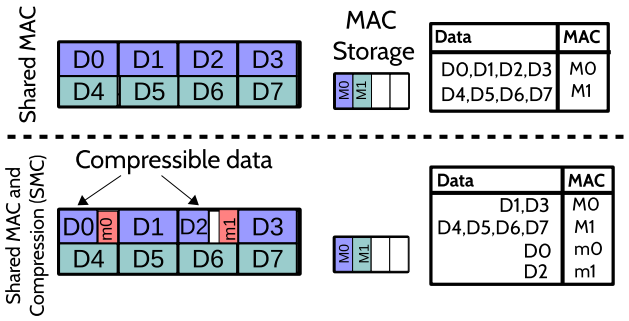

Figure 5. Shared MAC with Compression (SMC).

blocks can introduce (white) "holes" in memory that are not exploited for other uses.

Compression itself is a minor overhead relative to the cost of encryption and integrity verification. Recent compression algorithms, e.g., Base-Delta-Immediate (BDI [34]), are designed for simplicity instead of a high compression ratio. Note that in this context, we only require a block to be compressed by a factor of 1.14×. Prior work has shown that BDI compression/decompression can be implemented with a latency of 2 cycles and power of 33mW [36]. The compression and decompression are performed entirely in hardware and are transparent to the operating system.

3.4 On-Demand MAC Allocation (ODMA)

So far, we have employed sharing to mitigate the significant MAC capacity overhead. For further reduction, we propose to allocate MACs just for sensitive pages. To achieve this goal, instead of reserving a MAC region for the entire memory, we allow the OS to allocate a MAC entry for each sensitive page on-demand (ODMA). We include a pointer to the MAC location in the page's EPCM entry (extra 32 bits per 8KB page or 0.05% memory capacity). The delay introduced by the pointer indirection is incurred only on TLB misses. With this approach, the capacity overhead of MAC reduces linearly with the size of non-sensitive data.

Due to MAC allocation at page granularity in the eviction tree (Section 2.4), the SGX MAC overhead is trivial (i.e., 12 MB for EPC and 0.02% of the non-EPC region). The ODMA technique helps our scheme approach the low MAC overhead in SGX for applications with a small number of sensitive pages.

3.5 Security Analysis

The techniques introduced in this paper do not weaken security guarantees, relative to the baseline MEE algorithm. Techniques like VAUT and SMC continue to use similar sized hash functions as MEE to establish parent-child linkage and construct the MAC, respectively. Therefore, similar to MEE, for a replay attack to succeed, the attacker would

have to correctly guess the 64-bit output of the hash function or modify the inputs to the hash function such that it produces a hash known to the attacker, both of which have success probabilities of less than 2^{-64}.

A new operation in VAULT is the encryption used to generate the leaf nodes of the integrity tree, so we will focus on proving its security here. Since the hash function that generates the MAC for a data block is private to the CPU, the attacker must rely on a replay attack, i.e., the attacker must return an old block of data D, its corresponding MAC M, and the old counter value c that fulfils the relationship $M = hash(D, c)$. Any change to a non-leaf node of the tree will be detected by the integrity check in VAUT, exactly as in the baseline MEE. Therefore, to pull off a successful attack, the attacker must return a leaf node L', such that after decryption, the leaf L contains an old counter value c that the attacker can guess with high probability. The following encryption/decryption steps ensure that this is not possible.

For encryption, we use 128-bit AES. The plaintext leaf block L is first decomposed into four 128-bit sub-blocks L_0, L_1, L_2, and L_3. We create a new sub-block $L_x = L_0 \oplus L_1 \oplus L_2 \oplus L_3$, where $\oplus$ represents XOR. The sub-blocks, L_x, L_1, L_2, L_3 are then encrypted to create 128-bit sub-blocks for the encrypted leaf node L'. Each sub-block is created with the following encryption function: $L'_* = AES(L_*, P \oplus k)$, where k is the CPU's 128-bit private encryption key and P is constructed by concatenating the physical address (padded with zeroes to 52 bits) of the data sub-block and the corresponding 76-bit counter stored in the parent of the leaf node. During decryption, the reverse operations are performed: sub-blocks L'_x, L'_1, L'_2, L'_3 are decrypted to produce L_x, L_1, L_2, L_3; L_0 is then computed by performing $L_x \oplus L_1 \oplus L_2 \oplus L_3$.

With the above procedure, if the attacker returns a modified sub-block L'_*, it results in a modified decrypted sub-block L_*, and eventually a modified sub-block L_0. Since the attacker does not know the CPU's private key, from the perspective of the attacker, sub-block L_0 is a random unknown sub-block. As discussed above, to pull off a successful replay attack, the attacker has to correctly guess the 71-bit counter c; since the 64 global counter bits used to construct c are in random sub-block L_0, the probability of a successful attack is less than 2^{-64}.

Note that the encryption/decryption process has been constructed to ensure that a modified L'_* results in an L_* that is completely random from the perspective of the attacker. By XOR-ing the sub-blocks, we ensure that any modification to L' results in a random global counter value.

3.6 Discussion

Capacity Overhead

Table 1 summarizes the capacity overhead of various techniques. Note that SGX (Baseline) has an extremely low overhead, since EPC is a small portion of memory (96 MB) and the eviction tree works at page granularity (Section 2.4).

Type	MAC	Counter	Tree	Total
MT	0%	12.5%	14.2%	26.7%
BMT	12.5%	1.6%	0.8%	14.9%
SGX (Unified)	12.5%	12.5%	1.6%	26.6%
SGX (Baseline)	<0.3%	< 0.2%	≈ 0%	**0.5%**
VAULT	12.5%	1.6%	0.05%	14.1%
VAULT+SMC4	3.1%	1.6%	0.05%	**4.7%**
VAULT+SMC8	1.6%	1.6%	0.05%	**3.2%**

Table 1. Memory capacity overhead for different integrity techniques. Except for SGX (Baseline), other schemes use one unified tree for entire 16GB memory space.

Processor	
ISA	UltraSPARC III ISA
size and freq.	1-core, 3.2 GHz
ROB	64 entry
Fetch, Dispatch, Execute, and Retire	Maximum 4 per cycle
Cache Hierarchy	
L1 I-cache	32KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
L2 Cache	8MB/8-way, shared, 10-cycle
Protocol	Snooping MESI
Hash cache	32KB per core (default)
DRAM Parameters	
DDR3	Micron DDR3-1600 [20],
Baseline DRAM Configuration	1 Channel 8 Ranks/Channel 8 Banks/Rank
Mem. Capacity	16 GB
Mem. Frequency	800 MHz
Mem. Rd Queue	48 entries per channel
Mem. Wr Queue Size	48 entries per channel

Table 2. Simulator parameters.

Compression and encryption

Compression and encryption are frequently used together, e.g., in file systems such as NTFS [32], ZFS [13], and Apple's HFS [29]. Compression does represent a side-channel – if system behavior can be observed, an attacker can estimate

SPEC2k6				**NPB**			
Name	Comp.	WS	PF	Name	Comp.	WS	PF
GemsFDTD	99.99	3k	22K	bt	0.2	2.6k	513
libquantum	0.38	672	0.1	cg	10.26	9k	2k
mcf	98.87	12k	92K	ep	2.58	24	0
gromacs	53.88	48	0	lu	94.75	2.7k	346
milc	9.2	3.3k	29K	ua	72.32	4.2k	685
h264ref	99.77	72	0	is	0.26	1.08k	9.5
omnetpp	65.23	19k	18k	mg	76.59	15k	6.8k
astar	82.71	48	0	sp	0.25	2.7k	774
bzip2	40.20	216	0	**SPEC2k6**			
hmmer	2.02	48	0	sjeng	89.62	264	308
lbm	0.08	1k	6k	soplex	96.84	504	9.7

Table 3. Benchmark's specifications. Comp (Compressibility in percentage), WS (Working Set size in MB), and PF (average number of page faults in 50M instructions).

the compressibility of data. However, there are no known exploits for this side channel and it is currently not deemed to be a critical vulnerability [24]. If such leakage is deemed critical, compression could be performed at a coarse granularity, or with an element of randomness.

4 Methodology

To evaluate our techniques, we conduct cycle-accurate simulations with 21 workloads from two benchmark suites: SPEC2k6 [18] and NPB [4]. These benchmarks are described in Table 3, along with their working set size, the number of page faults per 50 million instructions, and compressibility with the Base-Delta-Immediate [34] algorithm. The compressibility is defined as the percentage of blocks that can be compressed to 56 bytes or less. We generate the memory traces for these workloads with Simics [11]; these traces are generated for 4 million memory accesses after fast-forwarding to the region of interest and warming up the caches. These traces are then fed into cycle accurate memory system simulations with USIMM [9]. Table 2 shows the assumed Simics and USIMM parameters.

We modify USIMM to implement MT, BMT, SIT, and our proposed techniques. Every CPU read and write request is converted to the appropriate set of data block, tree, and MAC reads and writes. USIMM is augmented with a 32 KB cache per core to save most recently accessed integrity tree nodes. Most of our results are normalized against a non-secure system, showing the overhead imposed by memory integrity verification schemes.

To analyze reset overheads from local counters, we ran week-long simulations with Simics in functional mode. We confirmed that the reset overheads had stabilized and our statistics were not polluted by the initial simulation phase where counters were being warmed up.

To measure the page fault rate, we ran our benchmarks for 50 billion instructions (including 2 billion instructions for warmup) using the PIN tool [27]. We consider 96 MB memory for the EPC with the clock algorithm [6] for its page replacement policy. We repeated this experiment for different numbers of enclaves in the EPC and resized the EPC share for each enclave, accordingly.

5 Results

5.1 Evaluation of VAULT

We start by comparing the behavior of VAULT, against that of MT, BMT, and SIT. To exclude the effect of page faults, these integrity trees are extended to cover the entire 16 GB memory space. Figure 6 shows execution time for these four cases for each benchmark, for an 8-core model, normalized against a non-secure baseline. In all cases, VAULT incurs a lower execution time overhead, proportional to the memory bandwidth overhead. As shown in Figure 6, BMT outperforms SIT, since its counters are 8× smaller, and hence more cacheable. For the 8-core model, VAULT reduces execution time by 34%, relative to SIT.

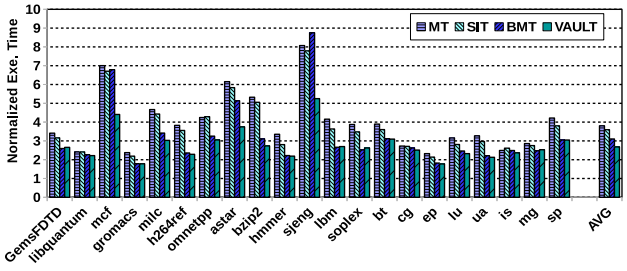

Figure 6. Execution time for MT, BMT, SIT, and VAULT, normalized against a non-secure 8-core baseline. All the trees cover the entire 16GB memory space.

The average breakdown of memory traffic for MT, BMT, SIT, and VAULT is shown in Figure 7. The integrity tree fetches are the dominant contributors in the baselines, but are sharply reduced for VAULT. The MAC fetch in VAULT is now a noticeable contributor, and is later targeted with our SMC approach.

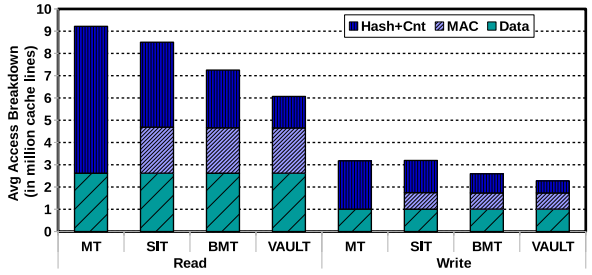

Figure 7. Average access breakdown for reads and writes in MT, BMT, SIT and VAULT.

5.2 Evaluation of Reset Overhead and VAULT

As we explained in Section 3, shrinking the sizes of counters might cause more resets. We ran separate long simulations with Simics in functional mode to analyze reset behavior for VAUT and VAULT, both with and without compression techniques. When using compression, there is one less bit per local counter because a bit is needed per block to store compressibility information. So the compression-based models are more susceptible to reset overheads. Figure 8 shows the 8 benchmarks most affected by reset handling. As shown in this graph, resets can incur an average overhead of 5.6% (up to 16% in one benchmark) in the VAUT+compression model. The VAULT organization is able to overcome the reset overheads. Even when using compression, VAULT has a reset overhead of only 2%. VAULT has to deal with decryption latency on the critical path. When assuming a decryption latency of 80 cycles [19], we see nearly zero impact in the 8-core bandwidth-constrained model.

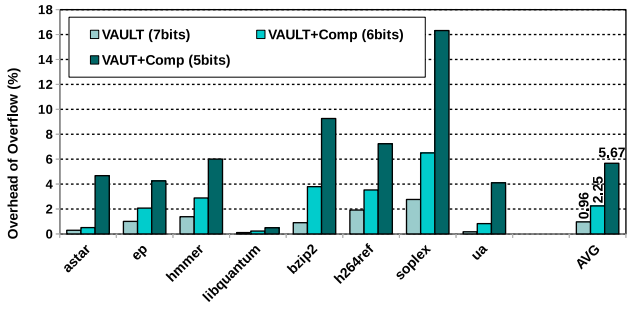

Figure 8. Execution time overhead introduced by counter reset handling. This graph only shows the 8 most affected benchmarks.

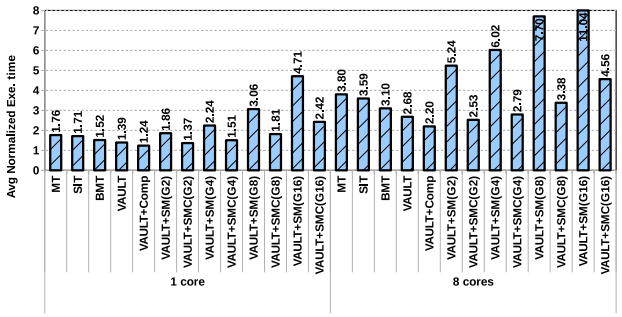

Figure 9. Average normalized execution time after applying the SMC technique with different group sizes, for varying core counts.

5.3 Evaluation of SMC

Figure 9 shows how execution time for SMC varies with group size. Recall that we are pursuing SMC to increase effective memory capacity, as summarized earlier in Table 1. We see that going from VAULT to VAULT+SMC with a group size of 1 reduces execution time by 21% in the 8-core case. This is because compression eliminates some MAC fetches. Since the group size is 1, i.e., no sharing, this model improves bandwidth and performance, but does not improve memory capacity. As group size is increased, the bandwidth penalty steadily increases, but has the side effect of growing memory capacity (not seen in this graph). Our experiments indicate that, in a single core system, sharing capacity overhead for group size of 1,2, 4, and 8 lead to 24%, 37%, 51%, and 81% performance overhead, with respect to the non-secure baseline, respectively. In other words, there is a capacity vs. performance trade-off in SMC. It is worth noting that on-demand MAC allocation can help us to use smaller group size to improve performance at a low capacity overhead.

5.4 Impact of Caching the Integrity Tree Nodes

We evaluate the impact of growing the sizes of the hash cache in Figure 10. We see that the hash cache shows steady

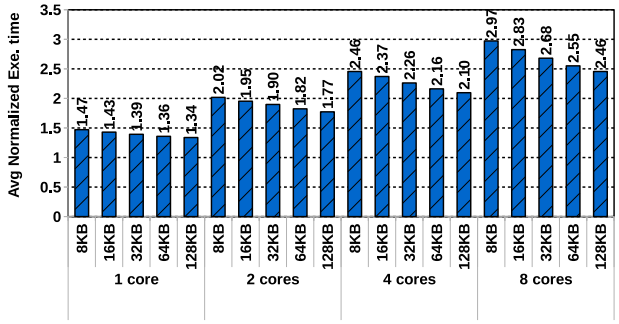

Figure 10. Normalized execution time as the size of hash cache changes from 8 KB to 128 KB per core.

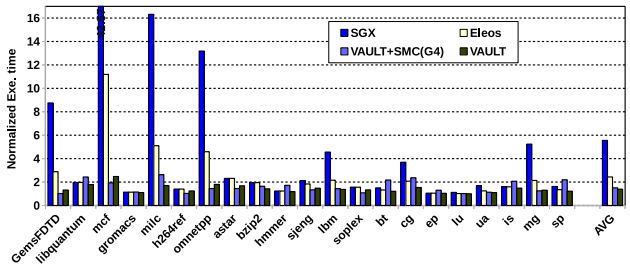

Figure 11. Execution time for SGX, Eleos, VAULT, and VAULT+SMC4, normalized against a non-secure 1-enclave system.

improvements in going from 8 KB to 32 KB to 128 KB. The improvement increases as the number of enclaves increases.

5.5 Page Fault Overhead

In contrast to VAULT, SGX suffers from page faults between the EPC and non-EPC regions. In this section, we evaluate the impact of page faults on SGX and the recently proposed software solution, Eleos [33], and compare them with our scheme. Eleos allocates two regions, one in the EPC region (called EPC++), and one in the non-EPC region (called backing store). It emulates the SGX model in these two regions at the software level, thus eliminating context-switches to the OS. Eleos moves pages from the backing store to the EPC++ before accessing them. Here, we consider an ideal case for Eleos. That is, we assume that every page fault can be resolved in the software layer and the overhead is just limited to the data transfer over the memory channel (8K cycles). For the baseline SGX, we consider 40K cycles per page fault [33]. Note that, as mentioned in Table 1, SGX and Eleos have negligible capacity overhead. Therefore, we also consider VAULT with SMC(G4), to make a fair comparison.

Figure 11 shows the slowdown of SGX, Eleos, VAULT, and VAULT+SMC4 for a single-core model, with respect to a non-secure system. Eleos outperforms SGX by 2.3× and VAULT+SMC4 outperforms Eleos by 1.61×. When the

number of enclaves increases (Figure 12), the performance gap also increases. More specifically, VAULT+SMC4 outperforms Eleos by 1.86×, 2.1×, 2.29×, for 2, 4, and 8 enclave models, respectively. The performance difference grows as data transfer for one core, on the shared memory channel, stalls the other cores' requests.

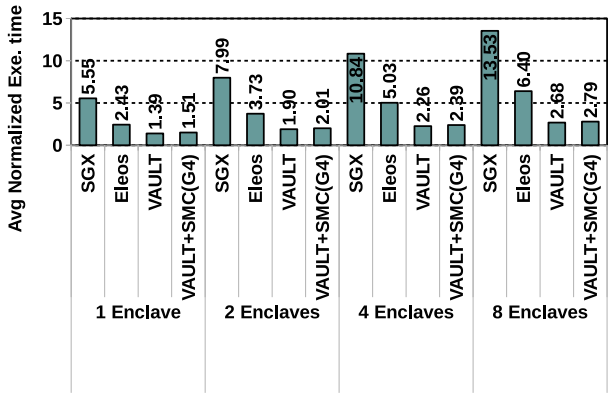

Figure 12. Average normalized execution time for SGX, Eleos, VAULT, and VAULT+SMC4 when the number of enclaves varies.

5.6 Summary of the Proposed Methods

In this section, we summarize the impact of each proposed method on the overheads of data integrity verification. Figure 13 captures the various design points and their trade-offs in terms of bandwidth and memory capacity overheads. Our workloads have an average working set size of 3.6 GB, so ODMA assumes that only 23% of all blocks are sensitive and need MACs. We see that the combination of the four proposed methods, VAULT, shared MAC, compression, and ODMA together, e.g., VAULT+ODMA+SMC4, yields the best performance with an affordable capacity overhead.

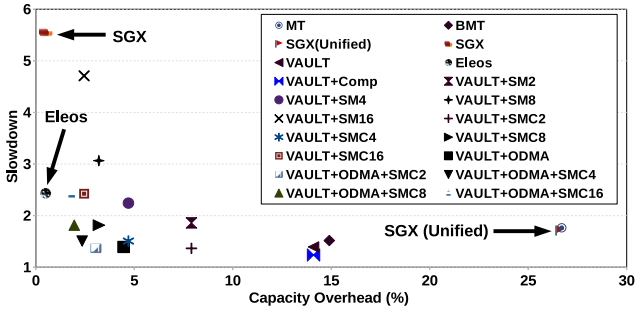

Figure 13. Comparison of different proposed methods.

6 Related Work

Lowering Merkle Tree Overheads. Several different schemes have been proposed to confirm memory integrity. Merkle Tree [31] was originally proposed to check signatures in public key crypto systems, it is also used for memory integrity verification [15]. Several studies have been conducted to decrease Merkle tree's overheads [14, 15, 23, 35, 37]. Gassend et al. [15] propose cached tree in which the hash values are cached on the CPU. Champagne et al. [7] reduce the size of the hash tree by excluding unused pages. Szefer et. al [23] employ a skewed tree that can prioritize the frequently accessed locations of memory by putting them in a leaf with a shorter path to root. Suh et. al [37] introduce Log Hash that checks for a sequence of accesses, to reduce verification checks. A few studies have designed integrity trees that can be updated and authenticated in parallel [14, 17]. Parallelizable Authentication Tree (PAT) [17] and Tamper-Evident Counter Tree (TEC-tree) [14] are two examples that update and authenticate data in parallel but with more capacity overhead than Merkle Tree. Recent work [1, 3] has shown that the overheads of integrity verification can be lowered by using smart memory devices.

Trusted Hardware Projects. Lie et. al [26] propose the eXecute Only Memory (XOM). XOM uses the combination of encryption, ownership tags, and MACs to protect its memory system. HIDE [42] addresses information leakage through the address bus, which was overlooked in XOM. Suh et al. [38] introduce AEGIS, an architecture for a single-chip processor that is able to work securely under both software and hardware attacks. AEGIS provides CIA protection for the main memory using encryption, MACs, and Merkle Tree. AISE [35] proposes a system resilient against hardware attacks and employs an address-independent encryption as well as Bonsai Merkle Tree. Bastion [8] proposes a hardware-software architecture to protect sensitive applications from other untrusted software. Flicker [30] provides a full isolated system for secure applications against any hardware and software attacks with a minimal TCB. PoisonIvy [25] is another architecture recently proposed to provide speculation for data without a significant overhead. Phantom [28] proposes a secure CPU that is able to obfuscate the addresses; moreover, this secure processor provides integrity verification by using Merkle tree and MAC generated by HMAC algorithm. SecureME [10] provides both hardware and software protection with a limited change to the OS. Similar to our work, SecureME considers a unified integrity tree. However, it does not follow the SGX programming model and it considers all the pages as sensitive for an application. None of the above architectures address the integrity tree *and* MAC capacity overheads.

Secure Solutions Based on SGX. Intel's Software Guard eXtensions (SGX) [12] is a set of extensions to the Intel architecture to provide confidentiality and integrity guarantees for any secure application. Built on this architecture, several works have been proposed. Haven [5] guarantees the confidentiality and integrity of a secure program while it is still in communication with untrusted applications.

SCONE [2] leverages asynchronous system calls to build a secure container with a small TCB and a lower overhead. Graphene [39] demystifies the myth that programmers need to make a significant change in a program to prepare it to run in an enclave. Eleos [33] accelerates SGX by managing the page faults in software level. Finally, HotCall [41] introduces a new interface for enclaves to mitigate the overhead of system calls.

7 Conclusions

SGX incurs a significant cost when it moves a sensitive page from the non-EPC region to the EPC. This work proposes extending the EPC to cover the entire physical memory while allowing the EPC to accommodate non-sensitive pages. However, naively growing the EPC leads to a large integrity tree and a significant capacity overhead for MACs. We introduce VAULT that takes advantage of split counters to increase integrity tree arity and reduce the intergity tree storage overhead from 12.5% to 1.6%. Furthermore, we use a combination of compression and MAC sharing to reduce the MAC capacity overhead from 12.5% to 3.2%. We observe that sharing a MAC across 4 or 8 cache lines represents a sweet spot. Finally, we show that allocating MACs just for sensitive pages can further reduce MAC overhead. The combination of all these proposals outperforms SGX by 3.7× while imposing a 4.7% capacity overhead, in a single-enclave model.

Acknowledgment

We thank the anonymous reviewers for many helpful suggestions and our shepherd, Rob Johnson, who pointed out a couple of vulnerabilities that we had not anticipated. This work was supported in parts by NSF grants CNS-1302663, CNS-1423583, and CNS-1718834.

References

[1] S. Aga and S. Narayanasamy. 2017. InvisiMem: Smart Memory for Trusted Computing. In *International Symposium on Computer Architecture*.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*. 689–703.

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

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1994. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications* 5, 3 (Fall 1994), 63–73. http://www.nas.nasa.gov/Software/NPB/

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[6] Richard W Carr and John L Hennessy. 1981. WSCLOCK: A Simple and Effective Algorithm for Virtual Memory Management. *ACM SIGOPS Operating Systems Review* 15, 5 (1981), 87–95.

[7] D. Champagne, Reouven ElbazRuby, and B. Lee. 2008. The Reduced Address Space (RAS) for Application Memory Authentication. In *Proceedings of ISC*.

[8] D. Champagne and R. Lee. 2010. Scalable Architectural Support for Trusted Software. In *Proceedings of HPCA*.

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

[10] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. 2011. SecureME: A Hardware-Software Approach to Full System Security. In *Proceedings of the international conference on Supercomputing*. ACM, 108–119.

[11] Wind Company. 2007. Wind River Simics Full System Simulator. (2007). http://www.windriver.com/products/simics/

[12] V. Costan and S. Devadas. 2016. Intel SGX Explained. (2016). https://eprint.iacr.org/2016/086.pdf.

[13] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). In *Request for Command (rfc)*.

[14] Reouven Elbaz, David Champagne, Ruby B. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. 2007. TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks. In *Proceedings of Cryptographic Hardware and Embedded Systems*.

[15] Blaise Gassend, G. Edward Suh, Dwaine E. Clarke, Marten van Dijk, and Srinivas Devadas. 2003. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), Anaheim, California, USA, February 8-12, 2003*.

[16] S. Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. In *Proceedings of IACR*.

[17] W. Hall and C. Jutla. 2005. Parallelizable Authentication Trees. In *Proceedings of SAC*.

[18] John L. Henning. 2005. SPEC CPU2006 Benchmark Descriptions. In *Proceedings of ACM SIGARCH Computer Architecture News*.

[19] R. Huang and G.E. Suh. 2010. IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability. In *Proceedings of ISCA*.

[20] Micron Technology Inc. 2006. DDR3 SDRAM Part MT41J256M8. (2006).

[21] J. Jeddeloh and B. Keeth. 2012. Hybrid Memory Cube – New DRAM Architecture Increases Density and Performance. In *Symposium on VLSI Technology*.

[22] J.Rodrigues, I.Torre, G.Fernandez, and M.Lopez-Coronado. 2013. Analysis of the Security and Privacy Requirements of Cloud-Based Electronic Health Records Systems. *Journal of medical internet research* 15 (2013).

[23] J.Szefer and S. Biedermann. 2014. Towards Fast Hardware Memory Integrity Checking with Skewed Merkle Trees. In *Proceedings of HASP*.

[24] J. Kelsey. 2002. Compression and Information Leakage of Plaintext. In *Fast Software Encryption*.

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

[26] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. (2000).

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

[28] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowic, and D. Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of CCS*.

[29] MACJournals. 2003. The HFS Primer. (2003). http://macjournals.com/~mwj/mwj_samples/MWJ_20030525.pdf

[30] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 315–328.

[31] Ralph C. Merkle. 1980. Protocols for Public Key Cryptosystems. *Security and Privacy, IEEE Symposium on* (1980).

[32] Microsoft. 2003. How NTFS Works. (2003). https://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx

[33] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys*. 238–253.

[34] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2012. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *Proceedings of PACT*.

[35] Brian Rogers, Siddhartha Chhabra, Yan Solihin, and Milos Prvulovic. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proceedings of MICRO*.

[36] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. 2014. MemZip: Exploiting Unconventional Benefits from Memory Compression. In *Proceedings of HPCA*.

[37] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*.

[38] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*.

[39] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC)*.

[40] M. N. Wegman and J. lawerence. 1981. New Hash Functions and Their Use in Authentication and Set Equality. *Journal of computer and system sciences* 22 (1981).

[41] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 81–93.

[42] X. Zhuang, T. Zhang, and S. Pande. 2004. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proceedings of ASPLOS*.