# LOW OVERHEAD SECURE SYSTEMS

by

Meysam Taassori

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2020

**The University of Utah Graduate School**

**STATEMENT OF DISSERTATION APPROVAL**

The dissertation of **Meysam Taassori**
has been approved by the following supervisory committee members:

| | | |
|---|---|---|
| **Rajeev Balasubramonian** , | Chair(s) | **24 Oct. 2020** Date Approved |
| **Erik L. Brunvand** , | Member | **24 Oct. 2020** Date Approved |
| **Mahdi Nazm Bojnordi** , | Member | **24 Oct. 2020** Date Approved |
| **FeiFei Li** , | Member | ___ Date Approved |
| **Mohit Tiwari** , | Member | **24 Oct. 2020** Date Approved |

by **Mary Hall** , Chair/Dean of
the Department/College/School of **School of Computing**
and by **David B. Kieda** , Dean of The Graduate School.

# ABSTRACT

Trusted Execution Environments (TEEs) allow users to store their data and outsource their computation over cloud servers without trusting cloud providers. The significant overhead of current implementations of security primitives is a major challenge for TEEs. In this dissertation, we propose several solutions to enhance the efficiency of a TEE while security guarantees stay intact or improve.

There are three major problems with the current implementations of security features: large metadata structures, data movement, and information leakage. In this dissertation, we propose several new structures for metadata and apply multiple techniques to address these three problems.

First, with shared counters and variable arity, we propose a smaller and more cacheable integrity tree to reduce its bandwidth and capacity overhead. Second, we introduce a more compact structure to store message authentication codes (MACs) and thus reduce its capacity overhead. These techniques enable a TEE to define secure pages across the external memory, while the metadata overheads are still in check. Therefore, pages do not have to be swapped between the secure and nonsecure regions, which alleviates the data movement overhead in current implementations. Third, we further improve the metadata overhead of the state-of-the-art system that uses a combination of integrity and error correction. We share the error correction metadata across multiple data blocks to reduce its footprint. This technique enables us to embed reliability metadata into the integrity tree to propose a single compact, unified metadata structure. This new metadata structure provides all required metadata blocks to support reliability and security guarantees, thus reducing the overall overhead for both.

Prior works share their security metadata structures among multiple applications, thus introducing a potential side channel. We address this issue by isolating applications such that each application has separate, isolated metadata structures. Not only does this technique eliminate the information leakage, it also improves performance due to the better

metadata cache efficiency.

In short, we thus propose $(i)$ a low-overhead TEE, which provides a scalable, secure memory for sensitive applications at more than $3\times$ lower bandwidth overhead relative to Intel® SGX, $(ii)$ low-overhead support for both reliability and integrity, which improves storage overhead by an order of magnitude and bandwidth overhead by more than 60% relative to the state-of-the-art technique, and $(iii)$ a leakage-free solution to provide integrity verification.

We thus confirm our hypothesis that by reducing the size of metadata structures, the overhead of security features will be reduced, while having little negative side-effects. We also demonstrate that by separating metadata structures across the applications, the potential side channel through these structures will be eliminated.

For my parents, Parvin and Mahmood,
and my brother, Mehdi.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

## 1.1 Trusted Execution Environments Challenges

Today, our computer systems are composed of millions of distributed devices at the front-end connected through the Internet to collect data and send it to the cloud servers at the backend to store or execute. Using the Internet as a backbone, Internet of Things (IoT) nodes inherit all issues of the Internet, including security and privacy concerns. Due to the ubiquitous nature of this network and its closeness to our lives, security and privacy threats become more critical in IoT nodes.

Furthermore, the growing demand for cloud services to store or outsource the computation over private data raises privacy and security issues. The statistics in recent years show that the innovations in cloud systems did not adequately address these issues. For example, The number of organizations hacked with a ransomware attack in 2019 has increased by 41% within one year [157].

Customers cannot trust cloud providers, nor other applications sharing the same platform with theirs. Even though cloud providers are spending an enormous amount of money on their security practices – Forrester reported that cloud providers are expected to spend $12.7bn on cloud security by 2023, which shows an 18% growth [5] – cloud customers are being continuously hacked; for example, in Cloud Hopper, hackers broke into cloud servers, including CGI and IBM, to steal a significant amount of intel® lectual property, security clearance, and other records from multiple companies over the past several years [42]. A survey by Cloud Security Alliance [31] – conducted every year in the past decade – shows that shared technology vulnerabilities, insider threats, and more importantly, the lack of a thorough security strategy are always in the top ten threats to cloud systems every year [14], [18], [26], [28].

The Cloudbleed buffer overrun vulnerability in Cloudflare in 2017, which affects more

than 3000 of its customers, is an example of the possible threat when multiple customers share one platform [23], [79].

A study [122] in 2017 shows that more than 58% of security breaches happened because of employee's behavior (i.e., insider threat), whether maliciously or not. The LinkedIn password hack [15], the MongoDB data loss on Amazon EC2 cloud [40], and the Yahoo data breach [20] are three examples of unintentional misbehavior of insiders, which leads to a substantial security threat; whereas, the Zynga data breach [70] is an example of a malicious employee who sold highly confidential business information to a third party.

In recent years, several terrifying attacks occurred due to the lack of a sound security strategy in big cloud providers, which compromised millions of users' records. The Facebook user data leak [112], and the Equifax data breach [138] fall into this category. The former happened because of insecure backups, which were available for public accesses without any authentication mechanism. The Equifax data breach was caused by multiple failures, some of which were not patching a well-known vulnerability in the system; not properly segmenting different parts of the system, which enable attackers to move from the web portal to other part of the servers and find usernames/passwords in plain-text; and finally, not renewing a certificate of one of the internal security tools, which allows attackers to exfiltarate customers' records without being inspected [73].

Besides vulnerabilities of cloud systems, an honest-but-curious cloud server may profit from customers' data by analyzing or selling their records. The Facebook-Cambridge Analytica data scandal [61] is an example of harvesting customers' profiles without consent for political purposes. Selling customer's query keywords by search engines to advertisement companies is another example of privacy invasion in cloud systems.

Because of cloud systems vulnerabilities – which cause the security accidents, and despite the effort cloud providers make to avoid them, these accidents continuously occur – *customers must consider cloud systems completely untrustworthy and independently try to preserve their privacy and security properties.*

Trusted Execution Environments (TEEs) allow users to store their data or outsource their code to execute on the cloud servers without trusting the cloud providers and other applications running on the same platform with theirs. Although TEEs seem to be a promising solution to use cloud services securely, their significant overhead imposed on

the applications discourages customers from requesting the security features.

Most commercial TEEs provide two main security features, confidentiality, and integrity, along with freshness [179]. Confidentiality is preserved by encrypting any data exiting from the CPU boundary. Integrity is the guarantee that a CPU read matches the data last written to that location. To verify data integrity, we append every 64B data block with a 64b hash tag named Message Authentication Code (MAC). The produced MAC is the output of a hash function with data block, the address, and a counter as its inputs.

To guarantee each data block's freshness, we associate it with a counter, which tracks the version of a data block. To further protect these counters from a replay attack, we maintain an integrity tree. Therefore, to verify a data block's integrity, besides the block, we are required to fetch one block of MAC, one block of counter, and multiple blocks of integrity tree nodes. Therefore, due to the additional memory accesses that must be made to provide integrity and confidentiality, it imposes a significant bandwidth overhead. For example, for a 16GB memory, we have to fetch 12 extra metadata blocks – one counter block, one block of MAC, and 10 blocks of integrity tree nodes (based on the Merkle tree [123]). Moreover, as mentioned, MAC values impose 12.5% capacity overhead.

To handle the significant overheads imposed to guarantee these properties, TEEs consider a limited size for their secure memory. If the working set size of an application exceeds the secure memory size, secure pages must be swapped between the secure and nonsecure regions of memory frequently. This page swap imposes a significant performance overhead because it requires an OS system call, context switching, page copying, and metadata updating.

Intel® SGX [7] is one of the commercial TEEs, which provides confidentiality and integrity. SGX splits the entire physical memory into two parts: the EPC and the non-EPC part. The EPC size is 96 MB – SGX reserves 128MB of memory called Processor Reserved Memory (PRM), 96MB of which is allocated to enclaves' code and data, and the remaining part is dedicated to metadata. SGX provides cache line-granularity metadata for data blocks as they are located in the EPC region, whereas it supports metadata at the granularity of page for secure pages when they are stored in the non-EPC region.

The "multigranularity metadata support" helps SGX keep the metadata overheads in check. However, as mentioned, secure pages must be swapped between these two regions

frequently during the execution time, which introduces the paging overhead. SGX suffers from three major sources of overhead [198]: (1) context switch, when an exception occurs or when an OS call is performed; (2) transferring data between the EPC and non-EPC regions, or paging overhead; and (3) secure memory access, which is imposed by MEE [82] to provide confidentiality and integrity against physical attacks.

The overall overhead of memory accesses, i.e., (2) and (3), that SGX imposes on sensitive applications is related to their working set size. If an application can fit into the LLC, the overhead is trivial. If its size is less than 96MB, the size of EPC, the overhead can go up to 12× for random memory access. Due to the paging cost, this overhead can dramatically increase to 1000× when the working set size exceeds the EPC size [36].

Besides this significant overhead, in current implementations of security properties, metadata structures are shared among different applications, thus introducing potential side channels. Microarchitectural attacks, such as Meltdown [110] and Spectre [99] have shown us that to exploit side or covert channels, physical access is not required. Therefore, it is more challenging to protect against this kind of attack.

## 1.2 Dissertation Overview

According to the prior discussion, we can draw a conclusion that the size of metadata structures maintained to provide a security feature plays a key role in its significant overhead; a large metadata structure can exacerbate the imbalance between CPU and memory speed, causing significant performance degradation. In this thesis, we address this considerable overhead by proposing more compact metadata structures. Smaller data structures need fewer memory accesses to fetch all required metadata. Moreover, the cachability of a compact structure is higher, increasing the cache hits, which results in reducing the number of memory accesses. In Chapter 4, we show the impact of a more compact metadata structure on the overhead of secure memory access, shown as (3) in Section 1.1.

We notice that by reducing the metadata overhead, the multigranularity metadata support, dictated by SGX, is no longer required. The size of the secure region in the external memory can grow while the bandwidth and storage overheads of metadata are still in check. Using this new compact metadata structures, In Chapter 4, we propose a low

overhead TEE built on top of Intel® SGX. We show that this new TEE can provide sensitive applications with *a scalable, secure memory* in which raising the working set size of an application does not increase the overhead of security features dramatically.

In Chapter 5, we show that the state-of-the-art technique that uses a combination of error-correction and security metadata to provide low overhead support for both, still suffers from large metadata structures, which causes the significant overheads. We share one block of error-correction metadata across multiple data blocks to reduce its footprint. This technique enables us to embed the reliability metadata in the security metadata structure to achieve a more compact metadata structure, which provides both, error-correction and integrity metadata, at a lower cost.

In Chapter 5, we also demonstrate that when metadata structures are shared among multiple applications, a side-channel attack is possible. Then, we propose a solution to eliminate the potential side channel, which improves performance as well.

### 1.2.1 Thesis Statement

We state that there are two major problems with the current implementations of integrity verification, the significant overhead due to the large metadata structures, and information leakage. Due to the wide gap between CPU and memory speeds, further pressure on memory bandwidth, caused by large metadata structures, can exacerbate the imbalance between CPUs and memory speed, imposing significant performance overhead. Moreover, in current security implementations, multiple applications share security data structures, and thus introduce potential side channels. In this dissertation we address these two issues.

*We hypothesize that the overhead of security algorithms can be reduced if we decrease the size of required metadata structures so that it is feasible to provide sensitive applications with a scalable, secure memory system. We further hypothesize that isolating applications by separating their metadata structures not only eliminates the potential side channel, it also improves performance.*

In the following section, we focus on the contributions of this thesis, where we provide a brief summary of VAULT (Section 1.2.2.1) and ITESP (Subsection 1.2.2.2). Finally, we describe the organization of this dissertation in Subsection 1.3.

### 1.2.2 Contributions

#### 1.2.2.1 VAULT: A Low Overhead Trusted Execution Environment

Intel® software guard extension (SGX) [21] provides confidentiality, integrity, and freshness for sensitive applications. To control the overheads of these security guarantees, SGX places the sensitive pages in a protected region of the memory, named *Processor Reserved Memory*, and makes its capacity limited to 128MB. If the sensitive working set of an application exceeds this size, sensitive pages have to be swapped between the protected, i.e., PRM, and unprotected regions frequently, which poses a significant paging overhead on the application's performance. To increase the PRM size to reduce the paging overhead, we need to overcome the large overhead of required metadata, which is proportionally increased by PRM's size.

The metadata overhead is classified into bandwidth and capacity overhead. Besides the data blocks, a secure system, e.g., SGX, has to fetch multiple metadata blocks to guarantee different security principles. These extra metadata blocks pose a significant bandwidth overhead on the memory system and make it much slower, widening the gap between the speed of the processor and the memory system. The second source of the metadata overhead is the capacity that these extra data blocks occupy. A secure system requires storing the metadata for future use, and due to its large capacity, they have to be stored in the memory system. For instance, more than one-fourth of the entire Processor Reserved Memory (PRM) is dedicated to the metadata to guarantee the promised security properties for the remaining region – which is called an *Enclave Page Cache (EPC)*.

As mentioned above, the main barrier to increase the PRM size to mitigate the paging overhead is the overhead of metadata, which grows proportionally – imagine that more than 25% of the entire memory system is unavailable to store data if SGX extends its PRM to the whole memory system. On the other hand, by increasing the protected region size, the number of metadata blocks required to protect one single data block increases, amplifying the bandwidth overhead.

We reduce the overhead of metadata significantly by proposing a series of techniques in VAULT. These techniques enable VAULT to allocate secure pages across the physical memory to eliminate paging overhead caused as secure pages swap between non-EPC and EPC regions in conventional SGX.

#### 1.2.2.2 Compact Leakage-Free Support for Integrity and Reliability

The current implementations of security algorithms have two major problems; significant metadata overhead, and shared metadata structures, which lead to information leakage. In this chapter, we mitigate these two problems by proposing *Isolated Tree with Embedded Shared Parity (ITESP)*.

Prior work shares a metadata structure among different applications, which introduces a potential side-channel – two adversaries can establish a covert channel. To mitigate this issue, we isolate each application by implementing separate metadata structures for each application. We show that not only does isolation eliminate the side channel, it also can improve performance.

Then, we share a metadata block among multiple data blocks to reduce the metadata footprint and lower the metadata bandwidth overhead. Sharing helps us design a more compact metadata structure and enables us to provide all required metadata blocks by accessing a single leakage-free structure, called ITESP, to improve performance and eliminate the potential side channel.

## 1.3 Thesis Organization

The rest of this dissertation is organized as follows. In Chapter 2, we provide some background on security and privacy concepts. Section 2.1 presents definitions of terms that will be used in this dissertation. Then, Section 2.2 gives a brief explanation about current Trusted Environments, and we focus on Intel® SGX, as our baseline, in more detail in subsection 2.2.4. Finally, we study different well-known attacks in Section 2.3.

Chapter 3 provides a thorough survey of the state-of-the-art techniques that try to implement different security features more efficiently. In Section 3.1, we study the proposals that provide integrity verification at lower overhead. Then we explore works supporting reliability (3.2) and reliability along with security (3.3). We also explain two works that exploit 3D memory to reduce the overhead of security properties (3.4). Finally, we elaborate on different studies trying to enhance SGX performance (3.5) and works that demonstrate various side-channel attacks mounted on SGX (3.6).

Chapter 4 proposes VAULT as a low overhead Trusted Execution Environment built upon intel® SGX. Chapter 5 introduces ITESP as a solution to address major issues of

providing integrity verification along with reliability. Finally, we conclude this thesis by summarizing the contributions and discussing new challenges as follow-up studies to this dissertation in Chapter 6.

# CHAPTER 2

# BACKGROUND

This chapter first clarifies some basic security concepts to provide an essential security glossary for future use in this dissertation. Then, we focus on academic and commercial trusted environments to explore their infrastructures, strengths, and weaknesses. Since in Chapters 4 and 5, Intel® SGX is considered as a baseline, we explain this commercial TEE in more detail.

## 2.1 Security Concepts

This section defines some basic concepts in the security field.

- **Confidentiality** is the protection of the secret's content from being disclosed to an unauthorized party.
- **Integrity** is the prevention of any type of data manipulation by an unauthorized party. When *Data Integrity* is guaranteed, it means that an authorized entity can read the data as it was written last.
- **Authentication** means to identify who a user or a system is. If authentication is provided, the system can ensure that requests are serviced to valid parties. Authentication without integrity is meaningless because authentication information should stay intact, and no attacker can manipulate it. Protecting sensitive data from any unauthorized accesses guarantees its confidentiality and integrity. Authentication is required to discriminate between unauthorized and authorized parties.
- **Availability** means providing a valid user with a particular service as requested. Availability ensures that no attacker can deny service to the users [179]
- **Freshness** guarantees that the last version of data is available. The validity of data does not suffice its integrity. It also should be guaranteed that data is "valid" and "fresh." To

provide this feature, *nonces* – numbers which are used once – contain the version of data blocks, thus guaranteeing their freshness [104], [179].

- **Obliviousness** is to hide, or obfuscate, the patterns of requests provided for an authorized party. By providing obliviousness, an attacker cannot obtain sensitive information, including the content of data or the users' identity, by observing the sequences of requests.

- **The Attack Surface** means the collection of ways – or different points of a system – through where an attacker can break a system's security. The wider the attack surface of a system is, the less secure and the more vulnerable it is.

- **Threat Model** clarifies the specifications of threats that are protected and defines the capabilities of attackers for a particular system. In other words, a threat model specifies the security features provided to protect a system and the vulnerabilities that cannot be protected.

- **Trusted Computing Base (TCB)** is a set of trustworthy hardware, software, and firmware components that designers rely on to provide security properties for the remaining parts of a system. In other words, the TCB is a set of components that no attacker can tamper with. The larger a TCB is, the more vulnerable the secure system is. That is why in today's secure systems, the operating system is excluded from the TCB; it is difficult to guarantee that in tons of lines of code in today's operating systems, there is no bug that attackers can exploit to compromise the system.

- **Secure Hash** is a function that maps an input ($m$), to a fixed-size output ($h$), called a hash value. The input size can vary, while the output size is fixed and smaller than that of the input. It should be mathematically impossible to reverse a hash function. It means that if $h = hash(m)$, it is computationally infeasible to calculate $m = hash^{-1}(h)$. Moreover, two inputs should not be mapped to the same output; it means that it should be difficult to find $m_1$ and $m_2$, where $m_1 \neq m_2$ and $hash(m_1) = hash(m_2)$. That is, a secure hash should be collision-resistant. Hash values are used to generate the Message Authentication Codes (MAC), provide digital signatures, derive a cryptographic key, and maintain integrity trees.

The purpose of MACs is to provide integrity and authentication together. To this end, it should be guaranteed that, unlike secure hashes, MACs can only be generated by designated parties, e.g., the owner. Therefore, to derive MACs, besides data, a cryptographic key is required, which is available to certain entities. If an attacker manipulates the data, he cannot generate the corresponding MAC because of the lack of access to the key. Therefore, a MAC tag can detect any malicious change to the data.

## 2.2 Trusted Execution Environment (TEE)

### 2.2.1 Introduction to TEEs

Trusted Execution Environments are protected environments, intended to preserve contents and states of data from different hardware/software attacks [150]. Based on their threat model, TEEs provide a secure environment for code and data to execute securely. TEEs usually provide confidentiality and integrity protection from hardware and software attacks.

In TEEs, confidentiality is enforced by "encryption," "isolation," and "flushing" [179]. If data emerges from the TCB – which is usually CPU package boundary – it has to be encrypted for the sake of confidentiality. Confidentiality against a software attack can be provided by isolation. A memory region allocated to a sensitive application can be isolated by owner checking the accesses, granting the owner's, and denying others' accesses. Authentication enables us to do owner checking. This isolation guarantees that only the owner can access its data, and hence the confidentiality of data is provided. States of an application can be secured by flushing the component containing the states as the application execution time ends. For example, when an application ends, before starting another application, registers, the cache system, and TLBs should be flushed. Flushing is required when the CPU contexts switching between two applications or threads.

### 2.2.2 Academic TEEs

There are a handful of academic architectures to establish a TEE to protect sensitive applications. eXecute-Only Memory(XOM) [108] assumes only CPU package is in the TCB, and the memory system is untrusted. XOM encrypts data blocks before storing them in the memory system. By allocating different keys to different applications to encrypt their data blocks, XOM implements isolation to protect confidentiality of sensitive data stored

in the memory system. XOM does not provide integrity. To fix the main weakness of XOM, AEGIS [177] provides confidentiality and integrity. Before writing a data block in memory, AEGIS encrypts data and appends it with a hash value, named MAC, to support confidentiality and protect data from unauthorized manipulations.

The Secret-Protection (SP) [105] is an architecture that uses hardware protection and some security mechanisms to establish a Trusted Software Module (TSM) to protect a sensitive application from software attack and a limited number of hardware attacks. SP encrypts data and uses MACs to provide confidentiality and integrity as data and code go to off-chip memory. To preserve confidentiality of registers and states, before switching to the operating system mode, SP encrypts the registers' content and appends them with MACs. Bastion [53], unlike SP, can establish multiple TSMs with different security domains in parallel. Besides encryption and MAC protection, Bastion provides the replay attack protection by maintaining an integrity tree for the memory system.

Beyond encryption, integrity protections, and replay attack protections, Ascend [72], [146] provides obliviousness to hide data access patterns. Ascend is also resilient against the differential power analysis techniques to conceal power consumption patterns as well. AISE [149] proposes a counter-mode based memory encryption to reduce the encryption overhead. AISE provides confidentiality, integrity protections, and replay attack protections at a lower overhead by introducing a novel integrity tree.

SecureMe [59] is a hardware-software mechanism that protects an application from physical attacks, malicious operating systems, and hypervisors. By using AISE [149], this trusted environment provides confidentiality and integrity over the entire memory system. SecureMe uses memory cloaking to ensure that a malicious OS or hypervisor does not have access to the plain-text data – operating systems and hypervisors do not require accessing the plain-text for their duties such as page allocation and swapping. SecureMe leverages a hardware mechanism in the secure processor, AISE, to do memory cloaking in order to reduce its overhead. Unlike SGX, SecureMe provides security features – integrity, confidentiality, memory cloaking – for entire memory. This technique defines two modes to access the main memory: the "cloaked mode" and "uncloaked mode." In the uncloaked mode, the cipher-text data is brought into the CPU, decrypted, and its freshness and integrity are verified; then, plain-text data will be passed to the application. In cloaked

mode, neither the integrity/freshness verification nor decryption is applied, the cipher-text data will be handed to the OS.

PoisonIvy [107] proposes a trusted environment that provides integrity verification, freshness, and confidentiality. This proposal uses speculation to improve performance of the integrity verification. PoisonIvy speculatively hands in plain-text data to the application without checking its integrity and freshness. Then, while the application consumes data, its integrity is also being checked. By running integrity verification in the background and passing the data to the CPU speculatively, PoisonIvy removes this process from the critical path to reduce integrity verification overheads.

Sanctum [63], SCONE [36], Ryoan [87], Haven [44], Eleos [140], SGXBounds [100], ZeroTrace [154], Graphene-SGX [185] have been proposed based on Intel® SGX. Sanctum tries to empower an SGX-enabled system to hide its page-level access pattern from a malicious operating system by making enclaves in charge of managing their own page table. By doing so, applications' page-level address patterns are invisible to the operating system.

SCONE accelerates SGX by implementing user-level threading to reduce the overhead of enclave transitions. SCONE also reduces the overhead of system calls in an SGX-based system by executing system calls outside the enclave. Ryoan uses SGX to establish a distributed sandbox. In Ryoan's threat model, neither enclaves nor infrastructures hosting enclaves are trusted. These two parties – enclaves and their hosts – try to steal users' secrets covertly, and Ryoan's responsibility is to detect and stop any covert channel between the infrastructure and the enclave.

Haven leverages an SGX-enabled CPU to implement shielded execution for unmodified applications which are not adapted for running in an enclave. Haven loads a Windows library OS along with the application inside an enclave; by doing so, a significant portion of the OS support is available inside the enclave. Therefore, Haven reduces the number of required system calls at the expense of increasing the TCB size. Eleos reduces the SGX overhead by handling the page faults in the software level. SGXBounds augments SGX with a low overhead memory safety approach. ZeroTrace provides obliviousness for an SGX-based system.

Similar to Haven, Graphene-SGX proposes a framework to run unmodified applica-

tions in an SGX enclave at a comparable performance overhead with modified applications. Graphene-SGX employs a library-OS, Graphene [184], to load an unmodified application into an SGX enclave. Unlike Haven, Graphene-SGX modifies the library-OS to run inside the enclave efficiently. Therefore, the TCB size in this framework is significantly smaller than that in Haven. Graphene-SGX and Haven support multitasking. Graphene-SGX instantiates a separate instance of the library-OS for each process inside the enclave. Unlike Haven, by doing so, this platform provides isolation for different processes [165].

Graphene-SGX concludes that by using a library OS along with several optimizations for SGX such as dynamic loading, it is possible to run an unmodified application in the enclave as efficiently as a modified application – and even more efficiently in some cases. Graphene-SGX tries to load a library OS in an SGX-based machine efficiently despite the fact that loading a library OS inside an enclave enlarges the TCB size significantly, thus increasing the SGX overhead. Tian et al. [183] show that this attempt has failed.

### 2.2.3 Commercial TEEs

#### 2.2.3.1 ARM TrustZone

The TrustZone architecture splits hardware and software resources into two parts, the *Secure world* for sensitive applications and the *Normal world* for nonsensitive applications. The TrustZone architecture guarantees that no Normal world components can access any Secure world resources. Moreover, in the TrustZone infrastructure, a single physical core can execute the Normal and Secure worlds' jobs securely in a time-sharing manner. By doing so, a TrustZone-based system does not require to dedicate one core to secure applications, thus saving power and area consumption [3], [25].

To separate these two worlds, TrustZone extends the address bus in the AMBA bus with an extra control signal, named "nonsecure bit (NS bit)." The NS bit indicates whether an access belongs to the Secure world or not. This new AMBA bus is called *AMBA3 AXI*. nonsecure bus managers set their NS bit to one to ensure that they cannot access any secure components. Setting the NS bit generates an address that does not match any secure components' address, guaranteeing the isolation between these two worlds. If a nonsecure bus manager tries to access a secure component, the bus or the secure component raises an error and halts this action.

TrustZone enables the CPU to communicate with peripherals securely. In AMBA3, there is a low power, low bandwidth bus for peripherals called *Advanced Peripheral Bus (APB)*. The APB connects to the system bus through a bridge, which is called *AMBA3 AXI-to-APB bridge* [136]. The APB bus does not have the NS bit for compatibility with peripherals. However, the AMBA3 AXI-to-APB bridge guarantees all security requirements for the TrustZone architecture. This bridge prevents nonsecure peripherals from accessing the Secure world, and it rejects unauthorized requests to secure peripherals as well.

Every physical CPU core in the TrustZone architecture can work as two virtual cores: Secure and nonsecure cores. The nonsecure core has access to only nonsecure components, while all resources are available to the secure core. The CPU core contexts switch between these two worlds in a time-sharing fashion to mimic these two virtual cores. Although this policy helps the architecture save power and silicon area, switching between these two worlds may open this architecture to a wide range of side-channel attacks [4], [102], [109], [135], [136], [170], [204].

To context switching robustly and securely between these two virtual cores, the TrustZone architecture proposes a new mode called "Monitor mode." The monitor mode implements the interprocess communication between the software of these two worlds. The Secure world's software can set its NS bit to jump into the Normal world. However, context switching from the Normal world to the secure one has to follow a couple of rules. This transition happens only through the monitor mode. A Normal world's software can switch to the monitor mode by executing an instruction, named Secure Monitor Call (SMC), or by hardware exception mechanisms – e.g., IRQ, external data abort, or external prefetch abort exceptions. The monitor mode stores the state of the source world and restores the state of the destination world. The monitor mode is a subset of the Secure world. The processor can access both secure and nonsecure addresses when it is in the monitor mode.

The TrustZone implements separate page tables for each virtual core to guarantee the isolation between the Secure and Normal words. This feature enables Secure and Normal worlds to have independent control over their virtual addresses and translations to physical addresses. Virtual page tables include the extra security bit, NS bit; the NS bit is used by the secure virtual processor to determine the world it intends to access. Whereas,

the nonsecure processor ignores this bit and considers its value as one to ensure that no unauthorized access to the Secure world can be made.

The TrustZone architecture supports two configurations for TLBs; if TLBs contain the NS bit in address tags, the monitor mode does not have to flush the TLBs when it context switches between two worlds, thus accelerating the context switching process. In the second configuration, TLBs do not include the NS bit, and hence, they have to be flushed when context switching happens [62]. Although the former configuration is faster than the latter, the latter is more secure, preventing information leakage between two Secure and Normal worlds.

The cache hierarchy in the TrustZone architecture can also be implemented in two configurations: first, the cache hierarchy contains the NS bit in the data tag to indicate each block's security state. In this configuration, each block of the cache can belong to the Secure or Normal worlds. This security state can change dynamically. Every block can be replaced by another one regardless of their security states. In the second configuration, blocks are agnostic to the NS bit. Instead, there is a component called the *TrustZone Memory Adapter (TZMA)* [3] to partition internal memories, i.e., SRAM or ROM, into two secure and nonsecure regions.

*The TrustZone Address Space Controller (TZASC)* is a configurable component in the TrustZone architecture, which is responsible for partitioning the address range of the corresponding node – the component which is attached to TSZASC – to define different regions. The TZASC is also in charge of partitioning the DRAM main memory in the TrustZone infrastructure. Therefore, in this structure, separate main memory systems are not required, leading to a lower cost, area, and power consumption. However, as always, shared resources among secure and nonsecure applications can make a system vulnerable to various types of information leakage. The size and number of regions can be configured on the TZASC.

The TrustZone architecture is not resilient against types of physical attacks. Considering CPU package boundary as the TCB for this architecture, while code and data reside inside package, they are not subject to physical attacks. TrustZone document [3] recommends keeping sensitive data and code inside CPU package, stored in on-chip memory. However, following this recommendation imposes a rigid restriction on the size of sensi-

tive code and data [62].

#### 2.2.3.2 Intel® SGX

Since in the entire dissertation, we pick an SGX-based system as our baseline, We explain this TEE in more detail in Section 2.2.4.

#### 2.2.3.3 AMD SEV

In 2016, AMD proposed the first version of its TEE, named *Secure Encrypted Virtualization (SEV)* [10], to isolate virtual machines (VMs) from the underlying hypervisor and other VMs. This technology enables cloud customers to run their VMs without trusting the cloud system and getting affected by potential bugs in the cloud infrastructure. SEV encrypts VMs' data stored in the main memory; therefore, the untrusted hypervisor, other VMs, or even an administrator can only access the encrypted version of VMs' data.

When context switching occurs, the hypervisor stores all VM's states, including registers' content in the main memory available to other untrusted hypervisors. To mitigate this issue, AMD introduced the second version of SEV, named *SEV with Encrypted State (SEV-ES)* [95]. Beyond the memory encryption, SEV-ES provides confidentiality and integrity for VM's registers to protect their states and contents from untrusted hypervisors.

Since to manage a VM, the hypervisor needs to access its registers state encryption can be challenging. To address this issue, in SEV-ES, the VM explicitly shares registers' state with the hypervisor. That is, VMs decide what information should be shared with the hypervisor [27]. This policy leaves the hypervisor in charge of managing the VMs, while the sensitive information of VMs is still intact and protected.

The next generation of SEV, called *SEV with Secure Nested Page (SEV-SNP)* [30], is built upon SEV and SEV-ES. SEV-SNP adds memory integrity protection on top of data and registers encryption to prevent unauthorized accesses to VMs' data. This technology protects VMs' sensitive information from replay attacks, memory aliasing, and memory remapping when a page is swapped out of the memory.

AMD SEV defines two types of pages, Private and Shared. The former contains sensitive information, while the latter's content is not confidential. To determine the type of pages, one bit, named *C-bit*, is stored in the page table entry. C-bit defines whether or not the pages' content should be encrypted.

In the SEV-SNP threat model, AMD is assumed to be trustworthy, the AMD SoC hardware and AMD secure processor are considered to be trusted. This threat model assumes that the VM is secure, trusted, and bug-free. The remaining part of the system – including BIOS, drivers, hypervisor, cloud software, and other VMs – is considered untrusted. In SEV and SEV-ES, the hypervisor was "benign but vulnerable," meaning that while the hypervisor is not fully trusted, it does not attack the VMs purposefully. In SEV-SNP, the threat model is stronger, assuming that hypervisor can be malicious too.

SEV-SNP, like its predecessors, can not protect the VMs against any physical attacks. Therefore, attacking the DRAM bus and manipulating data over the memory bus is beyond the scope of the SEV threat model. However, SEV[1] is resilient against cold boot attacks [30]. To provide integrity against the malicious hypervisor or other VMs, SEV-SNP ensures that the private pages are only accessible to their owners.

In the memory aliasing attack, one single physical page is maliciously mapped to multiple VM pages, which leads to memory corruption. SEV-SNP protects the VM's memory by ensuring that one physical page cannot be mapped to more than one VM's page simultaneously. A memory remapping attack means that the malicious hypervisor maps one VM's page to multiple physical pages. SEV-NP addresses this issue by ensuring that one VM's page is always mapped to one physical page; if this mapping needs to change, it should be validated by a trusted secure processor.

To do owner checking, SEV-SNP introduces a new data structure, called the *Reverse Map Table (RMP)* to keep track of the pages' owner. Combining with the page table, the Reverse Map Table enables SEV-SNP to enforce a couple of memory restrictions to assure that only the owner of private pages can access them. The RMP is a table indexed by physical address – in reverse of the page table, which is indexed by the virtual address.

AMD processors implement two-level paging, called *Nested paging* [1], to translate the VM's virtual address to the hypervisor's physical address. The guest page table first translates a guest virtual address (gVA) to a guest physical address (gPA). Then, the nested page table translates the gPA to the System Physical Address (SPA). Finally, one entry containing gVA and SPA will be created in the TLB.

---

[1] When it is written "SEV," it means all versions of SEV, including SEV, SEV-ES, and SEV-SNP.

In SEV-SNP, the RMP is indexed by SPA; therefore, to check the ownership, the SPA indexes the RMP; each RMP entry contains the corresponding gPA. If the entry's gPA matches the gPA translated in the nested page, the access is granted, and a new TLB will be created. Otherwise, the access will be denied, and a page fault will be generated.

As mentioned, since every RPM entry contains the gPA at which the corresponding physical page will be mapped, every SPA can only be mapped to one gPA, which means memory aliasing attacks can be detected. SEV-SNP also guarantees that mapping one gPA to multiple SPAs is forbidden – i.e., a memory remapping attack is impossible. Note that the nested page table assures that one gPA can map to one SPA. To prevent the untrusted hypervisor from manipulating the page table, SEV-SNP validates all changes made on the nested page table in a process called *Page Validation*.

Each RPM entry has a Validate bit, which is set to zero by the CPU as a page is allocated to a VM. As long as this bit is zero, the page can not be used by the VM. To validate a page, the VM has to confirm that it has not validated any other page with the same gPA. To do so, a VM needs to keep track of gPAs of the pages it has validated. This process guarantees that the memory remapping attack will be detected. After validating a page, the VM sets its validate bit to one.

The SEV-SNP document [30] states that this architecture is vulnerable to all types of side-channel attacks, or cache attacks, e.g., PRIME+PROBE. However, SEV-SNP is augmented with hardware capabilities to defend against some speculative side-channel attacks such as Spectre Variant 2 [99].

SEV-SNP has three indirect branch control features to mitigate the branch predictor based microarchitectural attacks; these features enable VMs to choose their policy in dealing with the indirect branch predictor [2]. AMD uses the"SPEC_CTRL" and "PRED_CMD" MSR to allow VMs to activate these features to have software control on the branch predictor structures. By activating the *Indirect branch prediction barrier (IBPB)* feature, the AMD processor guarantees that the older indirect branches cannot impact the prediction of the indirect branches in the future. This feature can be used when context switching between VMs occurs. The *Indirect Branch Restricted Speculation (IBRS)* feature can isolate indirect branches from different privilege levels – i.e., when "CPL=3" vs. "CPL= 0 to 2," or guest vs. host. If this feature is active, after transitioning to a higher privileged mode, indirect

branches cannot be influenced by branches in the less privileged mode, nor be controlled by branches of other logical processors. If the *Single Thread Indirect Branch Predictor (STIBP)* feature is active, indirect branches cannot impact the prediction of other sibling threads.

### 2.2.4 Software Guard Extensions (Intel® SGX)

Software Guard eXtensions (SGX) is a set of additional instructions to the Intel® processor ISA to provide security properties – i.e., confidentiality, integrity, and freshness – for sensitive applications [62]. SGX offers hardware-based memory protection, which isolates a region of memory, called *enclave*, from unauthorized accesses [13], [119]. SGX allows application developers to create an enclave over cloud servers to ship their sensitive code and data and execute software securely over cloud servers. SGX guarantees confidentiality and integrity, along with freshness, for data and code even in the presence of a malicious operation system (OS), hypervisor, or any applications running over cloud servers.

#### 2.2.4.1 Threat Model

SGX protects enclaves against a powerful adversary with privileged access to system software and hardware. The entire software stack, including the operating system, hypervisor, and other processes running on the same machine, are untrusted and might be compromised by an adversary. Furthermore, SGX can protect enclaves from compromised BIOS, drivers, System Management Mode (SMM), and Intel® Management Engine (ME). The Trusted Computing Base (TCB) in SGX covers processor package boundary and the attestation software, i.e., Quoting Enclave [12], [37]. Once data emerges from processor package, its confidentiality and integrity should be guaranteed. Since SGX assumes that the adversary has full access to the system hardware, i.e., the external memory, DMA, storage system, and the memory bus are untrusted; the adversary can probe the memory and storage buses [22].

Physical attacks targeting processor package are excluded from SGX threat model. SGX does not target different types of side-channel attacks, including power analysis attacks, side-channel attacks, and time channel attacks – e.g., cache timing attacks; moreover, the Denial Of Service (DOS) attack is also outside SGX threat model. It is worth mentioning that in SGX threat model, Intel® is assumed to be trustworthy, an Intel® chip is supposed

to work appropriately, and the private key is expected not to be compromised [43].

#### 2.2.4.2 SGX Memory Organization

A reserved region of the external memory protected by SGX from unauthorized accesses is called *Processor Reserved Memory (PRM)*. Data resident in this hardware-protected memory cannot be accessed by other software – e.g., operating systems, hypervisors, and even SMM – except for its owner. SGX also does not allow the DMA engine to transfer data into the PRM to protect this part of the memory from peripheral accesses. Intel® SGX implementation offers three PRM options that can be chosen in the BIOS: 32MB, 64MB, and 128MB. A subset of the PRM (about 96MB) is allocated to sensitive code and data, called the *Enclave Page Cache(EPC)*; the remaining part of the PRM is dedicated to metadata.

SGX assigns a range of virtual addresses, called *Enclave Linear Address Range (ELRANGE)*, to map to the enclave's code and data which reside in the EPC region. The virtual addresses outside the ELRANGE are assigned to nonsensitive pages located outside the EPC. ELRANGE is represented by a base (BASEADDRESS field) and a size (SIZE field) – two fields of the enclave's SECS. The size should be a power of 2, and the base should be aligned to the size. These restrictions help SGX simply check whether or not an address belongs to this range.

#### 2.2.4.3 Control Data Structures in SGX

SGX maintains several data structures to manage the EPC region and provide different security features for the code and data stored in this region. These data structures help SGX protect enclaves from malicious operating systems and applications. To understand how SGX provides the various security features, we briefly summarize some of the essential SGX data structures in this subsection.

- **The Enclave Page Cache Map (EPCM).** The EPC is partitioned into 4KB pages to store enclaves' sensitive data and code. To perform all security checks, SGX maintains some metadata for each EPC page in a data structure, called *Enclave Page Cache Map (EPCM)*. EPCM is an array in which every entry belongs to an EPC page. The fields of the EPCM is shown in Table 2.1 [62].

  PT in the EPCM structure shows the type of the corresponding EPC page. PT is "PT_REG"

**Table 2.1**: Fields in an EPCM entry

| Field | Bits | Description |
|---|---|---|
| VALID | 1 | "1" for allocated EPC pages |
| PT | 8 | Page type |
| ENCLAVESECS | - | indicates the slot number of the enclave's SECS, which identifies the page's owner |
| ADDRESS | 48 | Virtual address allocated to this EPC page |
| R | 1 | EPC page is readable |
| W | 1 | EPC page is writable |
| X | 1 | EPC page is executable |
| BLOCKED | 1 | indicates if the page is evicted |

for regular EPC pages storing enclaves' code and data, "PT_SECS" for EPC pages containing SECS information, "PT_TCS" for EPC pages saving thread control structures, or "PT_VA" for pages containing version arrays. "VALID" in the EPCM entry shows whether or not the corresponding EPC page is allocated to any enclave. "BLOCKED" indicates if the EPC page is evicted or not. Subsection 2.2.4.6 discusses how a combination of the BLOCKED and VALID fields can show the stage of an enclave. "ENCLAVESECS" is a field of EPCM that contains the slot number of the enclave's SECS, which is the owner of the EPC page.

Note that SGX does owner checking for EPC pages by checking the ENCLAVESECS field in the EPCM entry of the corresponding EPC page. By doing so, SGX can guarantee that each page is only accessible to its owner, and neither the system software nor other enclaves can access it. This owner-checking enables SGX to disallow an enclave to share EPC pages with other enclaves. However, enclaves can share pages in the non-EPC region.

- **The SGX Enclave Control Structure (SECS).** SGX also maintains metadata information per enclave to identify each enclave. This data structure is called *The SGX Enclave Control Structure (SECS)*, stored in EPC pages with the type of PT_SECS. The content of these pages is not accessible to enclaves or the system software; however, similar to other EPC pages, they can be evicted by the OS. In the enclave life cycle, allocating and deallocating an EPC page for the enclave's SECS are the first and last steps. Hence, this data structure is suitable for identifying enclaves. The system software uses the virtual address of the enclave's SECS to point to an enclave. Every SGX instruction gets the SECS virtual address as one of its inputs, and the system software also stores this information in page

table entries. Since the SECS structure is unavailable to enclaves and the OS, it contains secrets related to an enclave.

Two fields of "BASEADDR" and "SIZE" in the SECS represent ELRANGE of the enclave. "ATTRIBUTES" is another field of the SECS with ten subfields: The first subfield,"DEBUG," which shows whether or not the enclave is in the debugging mode – when the enclave is in the debugging mode, no security guarantee is provided. The second subfield is "XFRM," which is 64 bits and contains the value of the CPU's XCR0 during enclaves execution time. SGX guarantees that the value of XCR0 is equal to XFRM while an enclave is running. "MODE64BIT" is another subfield of the ATTRIBUTES field, which is set to one for a 64-bit enclave.

INIT, another subfield of ATTRIBUTE, shows if the enclave is initialized by `EINIT`. Once the SECS structure is created, this bit is unset, indicating that the enclave is not yet initialized, and its code cannot execute. Two other subfields, "PROVISIONKEY" and "EINITTOKEN_KEY," show if the enclave has access to Provisioning Key and Provisioning Seal and EINITTOKEN Key, respectively. The fourth bit and bits in the positions of six to sixty-three in the ATTRIBUTE field are reserved. The fourth bit must always be set to zero. "SSAFRAMESIZE" is another SECS field, which indicates the size of the SSA (explained later) in the number of EPC pages. It simplifies how to locate different fields of SSA.

The MRENCLAVE, MRSIGNER, ISVSVN, and ISVPRODID fields in the SECS structure enable an enclave to attest to the remote entities [24]. SGX maintains two registers, "MRENCLAVE" and "MRSIGNER," for attestation (Subsection 2.2.4.4) and sealing (Subsection 2.2.4.5), respectively. MRENCLAVE and MRSIGNER contain 256-bit hash digest, produced by the SHA-256 hash algorithm. MRENCLAVE, named Enclave Identity, is an internal log that records all the enclave activities after creation, including its code, data, and security flags. The second identity for enclaves, called "sealing identity," is stored in MRSIGNER and used for data protection. This register contains the hash value of the enclave author's public key.

Another SECS field, "ISVSVN," contains the enclave's security version number (SVN). The enclave author assigns an SVN to each version of the enclave. Different SVNs show

different security properties among various enclave versions. The "ISVPRODID" field contains the enclave product ID. Multiple enclaves that are authenticated with the same public key may end up with the same value in their MRSIGNER register. The Product ID field allows the author to differentiate enclaves with the same author's identity.

ENCLAVESECS is the physical address of the EPC page containing the corresponding SECS. Therefore, if this SECS page is swapped out of the EPC, the content of ENCLAVESECS is subject to change. Therefore, ENCLAVESECS is suitable for identifying an enclave as long as the SECS page resides in the EPC. SGX resolves this problem by using 64 reserved bits of the SECS structure to store an enclave ID, called *Enclave ID (EID)*. Some SGX instructions, which are involved with evicted pages, e.g., `EWB`, use EID to identify the owners of pages. The 32-bit MISCSELECT field in the SECS specifies which extended information should be stored in the SSA structure when an AEX occurs. For instance, if the least significant bit of MISCSELECT is 1, the exception information about page faults and general protection exception in the enclave is stored in this region [147]. A summary of the SECS's fields is shown in Table 2.2

- **The State Save Area (SSA).** When an exception occurs, SGX needs to store the thread's execution contexts in a data structure securely while the exception is being handled; this data structure is called *The State Save Area (SSA)*, which contains the values of general-purpose registers (GPRs) plus the future-specific registers, e.g., FP0-FP7 for FPU. Since

**Table 2.2**: Fields in the SECS structure

| Field | | Bits | Description |
|---|---|---|---|
| BASEADDR | | 64 | Base address to represent ELRANGE |
| SIZE | | 64 | the size of ELRANGE |
| ATTRIBUTES | DEBUG | 1 | set for debugging mode |
| | XFRM | 64 | the value of control register XCR0 during the enclave's execution time |
| | MODE64BIT | 1 | "1" for a 64-bit enclave |
| | INIT | 1 | Set once the enclave is initialized |
| | PROVISIONKEY | 1 | "1" when Provisioning Key is available from `EGETKEY`. |
| | EINITTOKENKEY | 1 | "1" when EINIT token key is available from `EGETKEY` |
| EID | | 64 | Enclave ID |
| SSAFRAMESIZE | | 32 | shows the size of SSA in number of EPC pages |
| MRENCLAVE | | 256 | a hash digest by SHA-256, contains Enclave identity |
| MRSIGNER | | 256 | a hash digest by SHA-256, contains Sealing Identity |
| ISVPRODID | | 256 | Product number for different modules of an enclave |
| ISVSVN | | 256 | Security version number for an enclave |
| MISCSELECT | | 32 | Specifies the extra features stored in the MISC region of the SSA as an AEX occurs |

this data structure is stored in a regular EPC page with the type of "PT_REG," the SSA's content is available to the owner enclave. The "SSAFRAMESIZE" field of the enclave's SECS contains the maximum number of EPC pages occupied by SSA data structures.

- **The Thread Control Structure (TCS).** Since SGX supports multicore processors, multiple threads may execute the same enclave code simultaneously. To manage the multithread execution, SGX uses a structure called *Thread Control Structure (TCS)* to store required information for different logical processors that execute the same enclave. EPC pages containing this data structure have the type of "PT_TCS" in their EPCM entries. These pages are not accessible to the system software or enclaves, including the owner.

#### 2.2.4.4 Attestation

Attestation is the mechanism of assuring a remote party that its code is hosted by a trusted container and executes securely. In this mechanism, the remote party is called *verifier*, and the trusted container is called *prover*. For the attestation process, SGX uses the *software attestation* mechanism, where the attestation process is implemented without any additional secure hardware at the prover side. The software attestation process employs a challenge-response protocol [34], where the verifier challenges the prover with a cryptographic signature. In other words, the remote party, a user, asks the prover, a cloud server, to sign a hash digest of initial states of the user's code and send the hash digest and the signature to the user. Then, the user verifies the signature with the public key provided by the trusted container. If the verification passes, the user can be convinced that the trusted container is trustworthy.

There are two types of attestation processes implemented in SGX; the first type is called *local attestation* or *intra-platform attestation mechanism*, which enables two enclaves executing on the same machine to authenticate each other. The second type is *Remote attestation* or *interplatform attestation mechanism* in which the prover and verifier are not on the same platform [33].

- **Local Attestation.** When two enclaves on the same machine are intended to cooperate, they need to authenticate each other in the first step. One enclave can prove its identity to another enclave, named target enclave, by calling the `EREPORT` instruction. This

instruction creates a signed structure, called REPORT. The REPORT structure contains the enclave measurements, i.e., MRENCLAVE and MRSIGNER.

The report structure is appended with a 128-bit MAC tag generated by the cipher-based MAC (CMAC) algorithm [67]. The `EREPORT` instruction uses a symmetric key, named *Report Key*, to produce the MAC. The Report Key is only shared between the `EREPORT` instruction and the target enclave. The target enclave can obtain its Report Key by running the `EGETKEY` instruction to recalculate the MAC tag and verify it. If the reproduced MAC matches, the target enclave can conclude that the sender is a valid enclave running on the same platform. The target enclave then produces a REPORT structure with the same MRENCLAVE field as the received one and sends it to the sender to confirm that it accepted the report message.

- **Remote Attestation.** Unlike the local attestation mechanism, the remote attestation process requires an asymmetric key. To do so, SGX creates an enclave, named the *Quoting Enclave* to handle the remote attestation process. The Quoting Enclave verifies the REPORT structure from another enclave, called attested enclave, in the same platform by using the local attestation method. Then, after verifying it, the Quoting Enclave replaces the MAC tag of the REPORT structure with a signature produced by a private asymmetric key to generate a message which is called a *QUOTE* [33].

  Three enclaves collaborate in the same platform to provide the remote challenger with an appropriate response: the first one is the *Attested enclave*, which is the application enclave intended to be attested; this enclave receives the challenge from the remote challenger. The second one is called *Provisioning Enclave*, which produces the encrypted attestation key for the third enclave, the *Quoting Enclave*. The Quoting enclave produces the final response and communicates with the remote challenger [62].

  Using the `EGETKEY` instruction, the Provisioning Enclave generates the *Provisioning Key* to authenticate itself to the Intel® provisioning service. When assuring that it communicates with an authenticated SGX processor, the Intel® provisioning service sends an *Attestation Key* to the Provisioning Enclave. Then this enclave uses the `EGETKEY` instruction to generate another key, called the *Provisioning Seal Key* to encrypt the Attestation key and store an encrypted version of that in the system software.

In the next step, the Attested enclave generates the local attestation report – similar to the report produced in the local attestation mechanism – and sends it to the Quoting Enclave. The Quoting Enclave uses the `EGETKEY` instruction to generate two types of keys. First, the Report Key to verify the attestation report received from the attested enclave. Second, this enclave obtains the Provisioning Seal Key to decrypt the Attestation key received from the Provisioning Enclave.

Finally, the Quoting Enclave replaces the MAC tag of the attestation report with the *Attestation Signature*, which is derived by the Attestation Key and contains a cryptographic hash of the enclave measurements and messages. The structure produced by the Quoting Enclave is called a *Quote*. The Quote is sent back to the remote challenger as a response.

The Quoting Enclave uses an asymmetric signing algorithm, in which the signer makes a signature by its private key, while the verifier can verify it using the signer's public key [62]. For this purpose, the Quoting Enclave uses the Attestation key received from the Provisioning Enclave.

#### 2.2.4.5 Sealing

The *Sealing process* enables an enclave to generate a seal key to encrypt its sensitive data and store it in the untrusted external memory when being destroyed. This process also allows an enclave to retrieve the key to decrypt its sensitive data once it is required. According to the way that `EGETKEY` generates a seal key, the enclave dictates how the sealed data can be accessed in the future.

Intel® has two policies to generate a seal key; in *Sealing to the Enclave Identity* or *Sealing to the Current Enclave*, `EGETKEY` uses the enclave's MRENCLAVE to produce the seal key. Therefore, the derived key is bound to the enclave's measurement. By choosing this policy any changes in the enclave content or version result in a different seal key. This policy is useful to assure that the old version of the enclave is not accessible anymore. *Sealing to the Sealing Identity* or *Sealing to the Enclave Author* forces `EGETKEY` to bind the seal key to the enclave's author, MRSIGNER, and the enclave's product ID (ISVPRODID). Hence, only an enclave with the same MRSIGNER and Product ID can unseal it.

Sealing to the enclave's author has two advantages over the previous policy; first,

different versions of an enclave can retrieve the same seal key. Second, authors can have the same seal key for all of their enclaves [22], [24].

The enclave invokes the EGETKEY instruction to generate the required seal key to perform the sealing process. After obtaining the seal key, the seal operation will be done using the AES-GCM encryption algorithm [153] to encrypt the data. It is essential to delete the seal key to prevent any unauthorized unsealing process. Finally, the sealed data, along with the key request structure, will be stored in the external memory. The unsealing process is straightforward. The seal key can be retrieved by calling the EGETKEY instruction. Then, the decryption process will be performed to unseal the data [24].

#### 2.2.4.6 Enclave's Life Cycle

An enclave passes through multiple states during its lifetime; these states are defined by the enclave's resources – especially EPC pages. In the SGX model, the system software manages the enclave's transitions between the states during its lifetime.

The system software creates an enclave by invoking the ECREATE instruction to allocate the first EPC page to the enclave SECS structure, initializing it by assigning the values to the SECS fields such as SIZE, BASEADDR, and INIT – INIT is set to zero. By calling ECREATE, the enclave will be in the "uninitialized" state. Then, the system software calls the EADD instruction to allocate new EPC pages and copy the data and code from the non-EPC region into these newly-allocated pages. EADD makes some security checks; it assures that the new EPC page has not already been allocated to another enclave – i.e., the VALID field on its EPCM should be zero. Moreover, The SECS of the owner of the new EPC page should not be in the initialized stage. Finally, EADD ensures that the assigned virtual address is in the ELRANGE range of the enclave. The system software uses the EEXTEND instruction to update the enclave's measurement while loading EPC pages.

At this step, the newly-created enclave cannot execute until it gets initialized. The system software calls the EINIT instruction to use a privileged enclave, named *Launch Enclave (LE)*. The Launch Enclave provides an EINIT token structure, EINITTOKEN, to initialize the uninitialized enclave. In the SGX system, each enclave should be vetted by a Launch Enclave before running the sensitive code. The LE approves the enclave by providing an EINIT token (EINITTOKEN). To issue the EINITTOKEN structure, the

LE requires access to the enclave's certificate signed by the author, named the Signature Structures (SIGSTRUCT).

First, EINIT verifies the SIGSTRUCT structure to initialize the SECS fields of the enclave and prepare it for execution. EINIT copies some SECS fields from the SIGSTRUCT structure and produces MRSIGNER. After verifying different fields of EINITTOKEN, EINIT sets the INIT bit to one, indicating the SECS is initialized [7]. Setting the INIT bit to one means that EADD cannot add any new pages to this enclave, and the enclave is allowed to execute its code. At this moment, the enclave is in the "Initialized not in use" state.

When a logical processor executes a code resident in the EPC pages, or inside the enclave, it is called the processor is *in enclave mode*. When the processor is *outside enclave mode*, it cannot make any accesses inside the enclave. Invoking the EENTER instruction, a logical processor becomes "in enclave mode," executing the code in the EPC pages.

A logical processor can exit from the "in enclave mode" by executing the EEXIT instruction, returning back to the nonsecure code. By executing EEXIT, control of the program will be transferred to outside the enclave. This form of exiting from the enclave mode is called *Synchronous Enclave Exit*.

If transferring to outside the enclave occurs because of a hardware exception such as a fault or an interrupt, it is called *Asynchronous Enclave Exit (AEX)*. By shifting the program control to outside the enclave, the enclave's state will change to the "Initialized Not in use" state. The EREMOVE instruction deallocates EPC pages and destructs the enclave. This instruction resets the VALID field of the corresponding EPCM entry to zero to free that page. Finally, EREMOVE deallocates the pages containing the enclave SECS to destroy the enclave completely.

#### 2.2.4.7 Paging in SGX

SGX leaves the untrusted system software in charge of paging. SGX does not change the address translation process, page allocation, or page table management. Instead, it adds security checks to monitor the system software to ensure that any malicious behavior is detected and halted. Similar to swapping a page from the main memory to the storage system, the operating system is allowed to pick a victim among EPC pages and evict it to the non-EPC region. However, SGX dictates a mechanism to guarantee that this process

takes place securely, and security features are provided for evicted pages while residing in the non-EPC region.

In regards to TLBs, two requirements should be met: First, SGX assures when a logic processor exits from an enclave, its TLBs are flushed. Second, when an EPC page gets deallocated from the enclave, SGX ensures that all logical processors involved with this page are forced to exit from enclave mode. Therefore, SGX guarantees that no TLB entry with the evicted EPC page resides in the TLBs. The system software sends an interprocessor interrupt (IPI) to cause a logical processor to exit from an enclave, which triggers an AEX and a TLB shootdown. To reduce the overhead of this process, the system software sends one IPI after evicting a batch of EPC pages.

To verify this process, SGX employs two bits in EPCM entries to define three different states for EPC pages: VALID and BLOCKED. An EPC page is *free*, if both bits are zero; it is *in Use*, if VALID=1 and BLOCKED=0, and the EPC page is *blocked*, if both bits are one. Requests targeting a blocked page will face a page fault. Therefore, while a page is blocked, its TLB entry is not going to be duplicated. Moreover, SGX instructions check to ensure that none of the pages they work with is blocked.

Before evicting a batch of EPC pages, the operating system runs the `EBLOCK` instruction for those EPC pages. This instruction sets the BLOCKED bit in the corresponding EPCM entries, which changes the state of these pages from "in Use" to "Blocked." This instruction ensures that no new translation targeting the corresponding TLB entries is responded.

Then, the operating system is supposed to remove the corresponding entries from the page tables. To ensure SGX that no malicious behavior is performed, after blocking the required EPC pages, the operating system issues the `ETRACK` instruction. This instruction allows SGX to keep track of logical processors that exited from the enclave, and their TLBs got flushed. Then, the operating system sends IPIs to logical processors to trigger an AEX and flush their TLBs.

Finally, to evict an EPC page into the non-EPC region, the system software executes `EWB`; this instruction takes a few steps to guarantee confidentiality, integrity, and freshness of an evicted page, while it is located outside the EPC. `EWB` requires a new data structure, called the *Version Array (VA)*, to store a nonce assigned to each evicted EPC page. `EWB` assigns an 8-byte counter as a nonce to each evicted EPC page to keep its version to

guarantee its freshness. This page version is stored in a Version Array (VA), which is an EPC page whose type is "PT_VA." Each VA contains 512 8-byte counters to keep the versions of 512 EPC pages.

Unlike regular EPC pages, VA pages are not allocated to any enclave, and hence, neither enclaves nor the system software has access to these pages. However, similar to other EPC pages, VA pages can be evicted by the system software. SGX needs to provide all essential security features for these pages while locating in the non-EPC region. To do so, SGX maintains a data structure, which is a forest of *Eviction Trees*; an eviction tree is a tree structure whose leaves are the EPC pages and nodes in upper levels are VA pages. An evicted VA page, along with its EPCM, is appended with a MAC tag and assigned with an 8-byte counter to keep its version. This assigned counter also needs to be stored in another VA page. These VA pages containing versions maintain the eviction tree to guarantee the freshness of the VA pages and, as a result, freshness of the EPC pages. The system software can shape these eviction trees because it doesn't have any impact on their security.

The `EWB` instruction appends the encrypted version of the EPC page, along with some fields of its EPCM, the owner's SECS, with a MAC tag, generated by the page version. The generated MAC, along with the metadata fields participating in the MAC generation, is stored in a data structure called *Page Crypto Metadata (PCMD)*. `EWB` writes all these structures except for the page version in the main memory; the page version will be inserted in an available slot in a VA page – each VA page has 512 slots for 512 pages' version. Note that every time one node of eviction trees gets updated, all its parents up to the root should be fetched, updated, and written back.

Although an evicted EPC page may still locate in the main memory, any address translation targeting it will face a page fault, the CPU control exits from enclave mode by an AEX, and the operating system invokes the page fault handler. First, if the evicted page resides in the storage system, it will be taken back to the main memory; then, the system software issues the `ELDB/ELDU` instruction to fetch an EPC page back to the EPC region and set the page's state. `ELDB/ELDU` verifies the MAC tag, produced by `EWB` at eviction time, and it only fetches the page if its MAC matches. `ELDB` sets the state of the newly-fetched page as "blocked" – i.e., BLOCKED=1, VALID=1, while `ELDU` tags it as "In Use" – i.e., BLOCKED=0, VALID=1.

#### 2.2.4.8 SGX Memory Access Protection

SGX is intended to protect the content of an enclave from hardware and software attacks. Hardware protection is performed by a component called Memory Encryption Engine (MEE), explained in Subsection 2.2.4.9. SGX protects against software attacks by a series of security checks. First, SGX guarantees that only the owner of each EPC page can access the page content; this guarantees confidentiality, integrity, and freshness of EPC pages against any software attacks.

Second, SGX needs to monitor the address translation process and checks a few security requirements to ensure that any malicious activity is detected and halted. These security checks mainly rely on the EPCM metadata and page table attributes.

SGX security checks a few page table attributes – it verifies the "W" (writable), "XD" (executable), and "S" (supervisor) flags in the corresponding page table entry to ensure that the access type and permission comply with the page table flags. If the logical processor is not in the enclave mode, any address translation is granted as long as the physical address is outside the PRM.

If the logical processor is in the enclave mode, SGX checks to ensure that all virtual addresses in ELRANGE are translated to physical addresses inside the EPC region and virtual addresses outside the ELRANGE range must be mapped to physical addresses outside the EPC region. Then, SGX verifies the ownership by comparing either the ENCLAVESECS field of the corresponding EPCM or the EID field in the SECS structure with the identifier of the access maker.

The next step, SGX checks the BLOCKED and the PT fields in the EPCM entry to ensure that the EPC page is not blocked, and the type is "PT-REG." As mentioned in Table 2.1, SGX maintains a reverse page table in the "ADDRESS" field of the page EPCM, which is the virtual address used to access the EPC page. The SGX security check will reject any given virtual address unmatched with this field [62]. If these security checks are met, this address translation will be granted to add to the TLB. The P, W, and XD flags in the page table are logically anded with the R, W, X flags in the EPCM entry.

SGX protects EPC pages from the active memory mapping attacks (Subsection 2.3.5). When an EPC page is allocated to an enclave, the assigned virtual address is saved in the "ADDRESS" field in the EPCM entry. Later on, at the address translation time, the CPU

ensures that the page table's virtual address matches the expected virtual address stored in the corresponding EPCM entry. SGX also defends against some passive memory mapping attacks (Subsection 2.3.5). SGX guarantees that access permissions of the EPC pages satisfy the owner's purposes. To this end, SGX stores the access permissions of EPC pages in their EPCM entries, i.e., R, W, and X in Table 2.1. When an EPC page is allocated, its access permissions in the page table will be overridden by the corresponding values in the page's EPCM entry. By doing so, SGX guarantees that the permissions entirely comply with the author's expectations. However, SGX is vulnerable to the passive address translation attack. We discuss SGX vulnerabilities against side-channel attacks in Subsection 3.6.

#### 2.2.4.9 SGX Memory Encryption Engine (MEE)

As mentioned in Subsection 2.2.4.1, SGX considers the processor package boundary as a part of the TCB, and thus the main memory and its bus are untrusted, subject to different attacks. To provide confidentiality, integrity, and freshness for memory accesses, SGX employs the *Memory Encryption Engine (MEE)* [82], which is a hardware component in the memory controller. The MEE can protect the main memory against a random corruption, replay attack, and cold boot attack; whereas, it does not hide memory access patterns or requests type.

EPC Memory requests must be routed through the MEE. For a write request, MEE encrypts a block of data, appends it with the 56-bit MAC tag, and assigns a 56-bit counter to keep track of its version. These counters guarantee data blocks' freshness and protect data against a replay attack. Likewise, for the read request, MEE computes the MAC tag and compares it with the received MAC to ensure that no unauthorized data manipulation occurs. If these two MAC tags do not match, the data block will be dropped, and the system will be halted [22]. If the MAC check is passed, the MEE decrypts the data block and stores the plaintext into the cache.

In the MEE threat model, adversaries have physical access to the machine. They can modify data blocks, write the plaintext of data blocks into any desired memory address, and fetch any data blocks from the main memory. They are able to store a block in the cache or evict any data blocks from the cache and send it to the main memory. Adversaries can also observe sequences of memory requests and tamper with the memory bus. However,

the processor package boundary is considered as a part of the TCB. Intel® is assumed to be trustworthy, and keys are supposed not to be forged [82].

The MEE maintains three keys, generated at booting time by the Intel® Digital Random Number Generator (DRNG) unit [6], which are kept in the MEE registers. These three secret keys are a 128-bit confidentiality key for encryption/decryption, a 128-bit Masking key, and a 512-bit universal hash key for integrity verification. The MEE uses the counter-mode based memory encryption; it splits a 512-bit data block into four 128-bit chunks and uses the AES128 algorithm to encrypt/decrypt these four parts separately. The MEE then concatenates these parts to create the ciphertext of the data block. To derive the crypto pad, the MEE uses the 128-bit "confidentiality key" and a seed composed of a 33-bit cache line address, a 2-bit index representing the data chunk location in the data block, and a 56-bit version counter. The MEE generates the ciphertext of the data chunk by Xoring the 128-bit produced pad and the corresponding data chunk – the one whose index is part of the seed. The MEE produces the encrypted version of data after repeating this process for four different parts of the data block and concatenating them together.

The MEE uses the Carter-Wegman MAC method [197] to produce required MAC tags. The MEE defines a 128-bit nonce composed of 33 bits of the data block address, and a 56-bit version of the data block, which is padded with 39-bit zeros. Using this nonce, a 128-bit masking key, and the AES128 algorithm, the MEE generates a 128-bit ciphertext of the nonce. Carter-Wegman MAC method employs the *multilinear universal hash* [196] in which the MAC tag is driven by $MACtag = h_{k_1}(data) \oplus f_{k_2}(nonce)$, where $k_1$ is a hash key, $h$ is a hash function, $f$ is an encryption function, e.g., AES, and $k_2$ is an encryption key [82].

The MEE breaks a 512-bit data block into eight 64-bit chunks $(Q_0, Q_1, ..., Q_7)$, and splits its 512-bit hash key into eight 64-bit keys $(k_0, k_1, ...., k_7)$. To generate the MAC tag, the MEE uses $MACtag = L \oplus Q_0.k_0 \oplus Q_1.k_1 \oplus ... \oplus Q_7.k_7$, where L is the 64-bit least significant part of the ciphertext of the nonce, and all operations are in $GF(2^{64})$. The 64-bit produced MAC tag will be truncated to 56 bits – 56 least significant bits are chosen – for the sake of the space concerns [82]. To guarantee the freshness of versions, the MEE maintains an integrity tree, which will be discussed in Subsection 4.2.4.

## 2.3 Attacks

In a modern computer system, the entire hardware and software stacks are subject to various types of attacks. These attacks can be classified from different aspects. In terms of the affected component, attacks can be divided into physical (or hardware) and software attacks; from the aspect of attack's impact on the system, attacks can be broken into passive and active attacks. In this section, we study different types of well-known attacks. In Chapter 3, we will discuss solutions to protect against some of these attacks. Knowledge about various types of attack helps engineers design a secure system which is resilient against a vast range of attacks.

### 2.3.1 Passive Attacks

Based on whether an attack may change the content of data, attacks can be divided into Passive and active attacks. In a passive attack, attackers do not alter data. They intend to obtain information about the victim, including the content of data or its footprint.

### 2.3.2 Active Attacks

In an active attack, data will not stay intact. The attacker tries to actively manipulate data, change the content, order, or version of data. Active attacks can be categorized as Spoofing, Splicing, and Replay attacks [104], [179]. In the memory-CPU model, *Spoofing Attack* is when an attacker can make unauthorized access to the block of memory by writing in the memory or injecting a read request. *Splicing attack* means that the contents of two or more memory blocks are exchanged. Another type of active attack is *Replay Attack*, an attack where the old version of a memory block replaces the current version.

### 2.3.3 Physical Attacks

Physical attack is the attack that an attacker exploits physical components of a victim to perform an unauthorized operation. To mount physical attacks, physical access to the victim is required. Whereas, a software attack can be mounted by executing software on a victim machine remotely. While the former is more expensive, complicated, harder to launch, and more challenging to detect, the latter is more common with a vast range of variety.

Some well-known physical attacks are as follows; *Port Attack* is a form of attack where

attackers exploit the existing ports of the computer to steal its secrets. One example is *Cold boot attack* in which the attacker uses the victim's USB port to boot it from the flush memory connected through the USB port; by doing so, the attacker can find access to the victim's peripherals. In one case of a cold boot attack, the attacker abuses the fact that the retrievable data remains in the DRAM memory for a while after power cuts off to dump the preboot DRAM data into a file.

*Bus tapping attacks* where attackers can tap the memory bus to sniff the memory data, actively change, or inject memory requests. Another form of physical attack is based on a correlation between the circuit power consumption and the computed data, which is called *Power Analysis Attack*. In this attack, attackers can infer the input value by measuring the power consumption.

### 2.3.4 Software Attacks

This class of attack can be performed by executing software on a victim machine. Mounting attacks on different parts of the software stack, including the operating system, hypervisor, and System Management Mode (SMM), falls in the software attack category. A compromised SMM – the most privileged level of software – allows the attacker to access all other parts of the software stack. One form of software attack is when attackers execute a malicious application on the CPU to compromise a physical component. Although in this form of attack, a physical component is compromised, in terms of cost and difficulty, this form of attack resembles the software attack [62].

*PCI Express Attack* is a form of software attack in which an attacker can exploit the PCIe bus to perform Direct Memory Access (DMA) to find access to the DRAM memory. Similar to the bus tapping attack, which is extremely expensive and difficult to launch, this method empowers an attacker to access DRAM directly at a lower cost relative to the bus tapping attacks. *DRAM Attacks* can be implemented through a piece of software code. A *Rowhammer attack* [80], [98] exploits the fact that by changing the content of a DRAM cell frequently, the charge of the neighbors is also subject to change. By implementing the Rowhammer attack, an attacker can flip the user/supervisor bit in the page table entry (PTE) to gain kernel privilege [160].

Some processors – such as Intel® Core Duo – are augmented with digital temperature

sensors whose data is accessible through the MSR register. Besides, the system software also has access to an array of performance monitoring events [147]. All this information enables a malicious OS to launch the *Performance Monitoring Side Channel attack*.

Motherboards have a flash memory chip containing their firmware, which is used to boot the machine. Although this approach makes updating the firmware more convenient, it also makes the system more vulnerable to a particular type of attack, called *Boot Firmware Attack*. An attacker tampers with the system software to inject malicious firmware and, therefore, modify the whole system software when it is loaded during the boot process. This attack may empower the attacker to access DRAM memory directly, similar to what might be obtained by the DRAM bus tapping at a lower cost and much more moderate difficulty.

### 2.3.5 Address Translation Attack

When an untrusted system software is in charge of address translation, managing the page table, and the page allocation process, a series of attacks is imminent. These attacks are known as the *Address translation Attack*. The system software requires to swap a victim page and replace it with a new one based on its replacement policy to perform the page allocation process. To this end, the system software relies on the accessed (A) and dirty (D) fields of the page table entries. By monitoring this information, applications' memory access patterns can be revealed to the untrusted system software. Since the attacker does not alter the content of pages, it can be known as a passive attack. Xu et al. [199] introduce a new class of attack, called *Controlled-channel attacks*, in which the operating system exploits the memory access pattern of an application to break the confidentiality of an SGX-enabled processor and access the plain-text version of encrypted images.

In another class of the address translation attack, the malicious system software tries to modify the page table to missteer an application to access the wrong page. Note in this attack, the attacker does not change the content of data directly; however, due to its impact on data, it is known as an active attack. Let us consider three scenarios, which cause active address translation attacks.

In the first scenario, a malicious system software modifies the page table such that virtual addresses are translated to wrong DRAM pages; this attack can fool the application

to access its data wrongly, which damages it significantly. In the second scenario, the system software tries to implement an active address translation attack without forging the page table. The malicious operating system can swap two pages into the storage system, exchange the contents, and swap those pages back into the DRAM memory. The contents of pages when they are swapped back are exchanged without altering the page table. Although the result of this active attack is as harmful as the first one, it is more challenging to detect this attack.

In the third scenario, a malicious system software does not invalidate the TLB entry of an evicted page, which leads to a security issue. Two pages are evicted from the DRAM memory while their corresponding TLB entries are kept valid. Then, when these pages are swapped back to the memory their locations in the main memory are exchanged, and the page table is updated accordingly; however, since the corresponding TLB entries did not get invalidated, the TLB entries do not get updated correctly and do not contain the last version of data. In the next address translation, there is a hit for these pages in the TLB, and the application will be misled to access the wrong page. The third scenario is harder to detect with the same result as the two first ones.

Section 2.2.4 explains that Intel® SGX can address all these active address translation attacks, while it is vulnerable to the passive one.

### 2.3.6 Cache Attacks

are one of the most potent side-channel and software attacks. In this attack, the attacker exploits the relation between memory access time and its location. In this class of attack, attackers can not directly access the victim's secret; however, they can obtain the victim's memory access pattern by measuring the access time.

*Prime+Probe* [111] is a cache attack technique that an attacker fills the cache with its data, the prime part, and then lets a victim access the cache. After the victim filled the cache with its data, the attacker again accesses all its data blocks. In this step, the attacker observes a longer latency to access one of its data block – this block got evicted because of the victim's memory access. By implementing this attack, the attacker can infer which address is touched by the victim.

*Flush+Reload* [201] is another way of mounting a cache attack that exploits shared pages between the victim and attacker to gain the victim's memory access pattern. In this scenario, the attacker flushes all shared data blocks from the cache. Then, it lets the victim access one of the shared blocks to store back into the cache. Finally, the attacker accesses all the shared blocks. The attacker observes lower latency in accessing the block fetched back by the victim compared with other blocks; this timing difference enables him to obtain the victim's memory footprint.

*Flush+Flush* [81], similar to Flush+Reload, this method also uses the shared data blocks between the victim and attacker to implement a cache attack. This method relies on the fact that the execution time of the `clflush` instruction depends on whether or not the target resides in the cache. In case of a cache hit, the block needs to be evicted from different cache hierarchy levels, whereas `clflush` aborts quickly when flushing a missing block.

To exploit this fact, the attacker flushes shared data blocks from the cache and lets the victim make access. Then, the attacker executes the `clflush` instruction on the same blocks for the second time, while measuring the execution time of the `clflush` instruction. Based on the measured time, the attacker can obtain the address victim had touched previously.

In contrast to other cache attack methods, this technique does not make any memory accesses, and hence its impact on the cache is minimum – it does not make any cache miss and makes only a few cache hits. Compared with Flush+Reload, this technique is less accurate, but faster with higher covert channel bandwidth.

### 2.3.7 DRAM Timing Side-Channel Attack

Every shared resource is prone to side-channel attacks. Similar to the cache hierarchy, there is a dependency between the DRAM memory access time and the location the access targets. Modern DRAM memory systems are organized in multiple channels, DIMMs, Ranks, Banks, and each bank contains multiple rows. The timing parameters in DRAM memory dictate different memory access times when accesses target various memory organizations' locations. An attacker can exploit this time difference to establish a covert channel.

DRAMA [144] exploits the DRAM row buffer, which is a shared resource in the mul-

tiprocessor systems to establish a covert channel. DRAMA uses the timing difference between memory access to an open row buffer, and a closed one to develop a covert channel. DRAMA also exerts this timing side-channel to obtain the victim's memory access pattern. To build a covet channel, the sender and receiver – both are malicious – make memory accesses on the same bank. The receiver keeps accessing while measuring the access time. To send one bit "one," the sender accesses the same row in the same bank where the receiver is accessing. Since the receiver's access leads to a row buffer hit – due to the sender's access, the receiver observes lower memory access time, inferring it as one bit "one." If the sender accesses a different row on the same bank with the receiver, then a row conflict occurs, increasing the memory access time experienced by the receiver, which can be inferred as one bit "zero."

Using different memory access latency caused by row conflicts, DRAMA [144] also performs a side-channel attack. To employ the timing leakage to infer whether or not the victim accesses a particular address (say $\alpha$), the attacker accesses two memory locations; one location in a same row of the same bank with $\alpha$, say $\beta$, and another one mapped to the different row of the same bank with the victim's address, say $\gamma$. Then the attacker makes a row conflict by accessing address $\gamma$ and waiting for the victim to make access. Finally, the attacker accesses address $\beta$ and measures the access time. If the victim accesses address $\alpha$, then there is a row buffer hit as the attacker accesses address $\beta$, and the attacker observes a lower latency in its measurement. Otherwise, there is a row conflict, which leads to a higher latency experienced by the attacker. therefore the attacker can infer the victim's footprint.

Since DRAM memory is a shared resource in multiprocessor systems – even if two threads do not access a shared page on the DRAM memory – one thread's accesses can affect the memory access time observed by another thread when these two threads access memory concurrently [191]. Two malicious threads can establish another covert channel by exerting this timing effect. To send one bit "one," the sender requires to access memory more frequently, which leads to an increase in memory access time observed by the receiver. Similarly, if the sender accesses memory less frequently, it occupies less memory bandwidth; therefore, the receiver can experience a lower memory access latency.

Wang et al. [191] and Shafiee et al. [162] try to eliminate this information leakage by

forcing the applications to make memory access at a constant rate. Camouflage [205] proposes a flexible approach to shape the memory requests; this proposal provides a trade-off between performance and security to allow designers to choose the amount of information, which is preserved and the performance overhead which the system should tolerate. Ferraiuolo et al. [71] consider a system composed of secure and nonsecure applications in which nonsecure applications do not require timing-channel protection. This assumption does meet reality and allow the memory controller to allocate the memory bandwidth more efficiently, reducing the overhead of timing-channel protection significantly. Using a lattice model, this work proposes a new scheduling algorithm that meets all security requirements of all entities in the system at the lower performance overhead.

### 2.3.8 DRAM Access Pattern Attack

Memory access pattern is a source of critical information whose leak can reveal the identity of a party [114]. Some studies [48], [103], [114], [180] have performed experiments where revealing the applications' footprint empowers the attacker to infer the secrets. Genome sequencing [48] and variant calling [180] are two famous applications in the genomic analysis, which require to access a hash table containing a reference genome. Brasser et al. [48] and Taassori et al. [180] have shown that if an attacker can observe the memory access pattern of these applications, the customers' genome will be revealed to the attacker and thus jeopardizing customers' privacy.

Membuster [103] demonstrates that an SGX-based system is also vulnerable to this class of attack. Membuster runs two applications – Hunspell and Memcached – in enclaves to show that observing the memory access pattern can reveal the enclave's secret. Hunspell uses a hash table to do spell checking. If an attacker can monitor this application's memory footprint, he can obtain the hash table entries accessed by Hunspell, and therefore infer the word, which is being looked up. In another experiment, Membuster executes Memchached as another victim in which secret is the data being searched in the Memcached cache. Similar to the first tool, there is a hash table in this database that the accessed entries address can reveal the secret.

To address this problem, Ascend [72] and Phantom [114] employ an Oblivious RAM (ORAM) infrastructure to hide the footprint of applications. ZeroTrace [154] implements

ORAM for enclaves in an SGX-based system to augment SGX with obliviousness. Oblix [127] uses multiple oblivious access techniques along with an SGX-based platform to design an efficient oblivious search index.

### 2.3.9 Denial of Service Attack (DoS)

A DoS attack occurs when a legitimate user can not use a resource because of unavailability due to the malicious activity of an untrusted party [11]. Moscibroda et al. [129] demonstrate that it is feasible to mount a DoS attack on multicore processors. Since current memory controllers are not informed of the owner of requests, one thread can abuse unfairness in memory scheduling policies and hog the entire memory bandwidth, posing a long memory latency to other threads. *Distributed DoS Attacks (DDoS)* occur when a node in a network maliciously generates a flood of requests to disrupt the normal traffic.

### 2.3.10 Man-in-the-Middle Attack

In this class of attacks, the attacker intercepts the communication between two or multiple trusted parties to interfere in their communication. The attacker can just sniff and not alter the data – which is a passive attack – or inject, steal, or forge transmitted data – which is an active attack [179].

### 2.3.11 Iago Attack

is a class of attacks that an untrusted operating system compromises an isolated and protected application by manipulating system call returns. For example, if an application uses the `getpid()` and `time()` system calls to obtain a seed nonce to generate a new random number – this scenario happens when in the SSL protocol, the server and client establish shared cryptographic secrets based on a public nonce. A malicious OS can implement a replay attack by replying to these system calls with compromised values [55].

# CHAPTER 3

# RELATED WORK

In this chapter, we provide a brief survey on the studies mitigating the large overhead of integrity verification algorithms (Section 3.1). We briefly look at a large number of studies which try to address DRAM memory reliability (Section 3.2) and attempt to provide both reliability and integrity efficiently (Section 3.3). Then, we continue with the impact of smart memories in secure systems (Section 3.4). In Subsection 2.2.4, we elaborated on different aspects of Intel® SGX. In this chapter, we discuss SGX performance improvement (Section 3.5) and SGX information leakage vulnerabilities (Section 3.6).

## 3.1 Memory Integrity Verification

One of the applications of hash functions is to generate a hash digest to detect any unauthorized modification on data. To ensure that only authorized parties can generate hash tags, they are bound to a cryptographic key; the hash function whose input is a data block along with a key is called *Message Authentication code (MAC)* function [104]. In memory systems, to assure that the MAC for different blocks in different addresses cannot be exchanged MAC function receives the memory address as another input – this ensures that splicing attacks are detected. Moreover, to protect against a replay attack, the MAC value should be bound to a version – or a counter [149] containing the data block's version. Therefore, MAC tag can be derived by $MAC\_tag = MAC(M, Addr, count, Key)$, where $M$ is a data block, $Addr$ is its memory address, $count$ is its version, and $Key$ is the cryptographic key.

A hash tree is required to protect against a replay attack. A hash tree is a data structure where leaves are the hash values for data blocks, and each node of the tree is the hash value of its children. Therefore, the intermediate nodes in the hash tree also contain hash values. The root is a representative of the whole data stored in the secure memory – located in the CPU package. The hash tree was initially proposed to check signatures in public-key

cryptography systems; the tree is called *Merkle Tree [123]*. Then, Gassend et al. [75] employ this tree to protect against replay attacks.

For data integrity verification, the corresponding hash values from the leaf up to the root in the Merkle tree are required. To write a memory block, we need to fetch all corresponding hash values, update them, and write them back to the memory. The Merkle tree has a large bandwidth and capacity overhead, so several studies have been conducted to mitigate its overheads [68], [75], [94], [149], [176].

Gassend et al. [75] propose a cached tree in which the hash values are cached on the CPU. In this scenario, to verify a data block's integrity, we just need to fetch hash values from the leaf up to the node of the tree that is resident in the cache. MAPS [106] analyzes different caching strategies for integrity metadata. MAPS observes that caching all types of metadata increases cache efficiency; this work claims that in the metadata cache, reuse distances are either long or short and always highly related to the type of metadata. MAPS makes an observation that traditional eviction policies are not useful for a metadata cache containing different metadata types.

Champagne et al. [52] reduce the size of the hash tree by excluding unused pages. Szefer et al. [94] employ a skewed tree that can prioritize the frequently accessed locations of memory by putting them in a leaf with a shorter path to the root. Suh et al. [176] introduce "Log Hash" that checks the integrity for a sequence of accesses, to reduce the performance overhead of this security property. In this technique, the CPU maintains read and write logs and updates them for future use. To update these logs efficiently, Log Hash uses incremental multiset hash functions [60].

A few studies have designed integrity trees that can be updated and authenticated in parallel [68], [83]. Parallelizable Authentication Tree (PAT) [83] and Tamper-Evident Counter Tree (TEC-tree) [68] are two examples that update and authenticate data in parallel but with more capacity overhead than Merkle Tree.

In Chapters 4 and 5, we will discuss the state-of-the-art integrity verification techniques – Merkle tree [75], [123] (Subsection 4.2.2), Bonsai Merkle Tree [149] (Subsection 4.2.3), MEE integrity tree [82] (Subsection 4.2.4), and Morphable Counters [151] (Subsection 5.2.2).

In the database literature, there are handful techniques to provide integrity verification for outsourced sensitive database systems [65], [118], [134], [142], [166]. Materla et al. [118]

and Pang et al. [142] employ authenticated data structures to assure clients of the authenticity and integrity of query replies. The drawback of these techniques is their significant bandwidth and capacity overheads. To address this issue, a group of work [131], [134] uses the signature aggregation mechanism to guarantee the correctness [131] or both correctness and completeness of query answers [134]. When multiple parties sign a document, signature chaining preserves the orders of signatures, guarantees the liability of signers, integrity, and authenticity of the document [35], [155], [156].

Blockchain is deployed to provide integrity guarantees in a distributed database [64], [74], [178]. Each block in a blockchain maintains the hash value of its previous block to create a linkage between every two consecutive blocks and a Merkle tree of a bunch of transactions to verify modifications. This structure enables blocks to validate each transaction to provide integrity supports [64]. Using a blockchain-based platform, a group of work [39], [46], [64] proposes low overhead solutions to manage healthcare databases and patients' sensitive information.

## 3.2 Memory Reliability

There are two possibilities when there is a mismatch between what the processor reads and what was last written. First, a failure in the memory system that corrupts the data – addressed by "Reliability solutions"; second, a malicious activity intentionally alters the data – detected and protected by "Security solutions" . Although the impact on the data is similar, when there is a combination of smartness and maliciousness behind the change, it requires a smarter workaround. Another similarity is that protection against both issues imposes a significant performance, capacity, and energy overhead.

To address the capacity overhead, Frugal ECC [97] and COP [141] compress a data block at cache granularity to store ECC metadata along with the data block. While both techniques provide ECC protection for a non-ECC DIMM, Frugal ECC can provide chipkill protection. Sharing one block of ECC metadata among multiple data blocks is another way to reduce the capacity overhead of reliability mechanisms. ECC-parity [92] shares the ECC bits among different channels to reduce the power and capacity overhead of a reliable memory system. Multi-ECC [91] provides chipkill for an ECC DIMM by leveraging a shared checksum.

There are several new challenges to provide reliability for 3D DRAM memories; this new DRAM family requires a new failure model because patterns of failure in 3D stacked chips are different compared with that in the traditional DRAM memory – e.g., due to the vertical structure, the outer dies can shield the inner ones from alpha particles, leading to a heterogeneous error rate across different layers [195]. Moreover, there is a new component, Through Silicon Via (TSV), in the stacked DRAM whose potential failure should be considered. It is well-known that the error rate in 3D DRAM memories is higher than that in 2D memory systems because of their higher density. Multiple studies [57], [90], [117], [133] try to provide reliability for the stacked memories at an affordable cost.

Citadel [133] provides reliability for 3D stacked memories when a large granularity failure occurs – which is commonplace in this memory family. Protecting against large granularity failures – like what we have in ChipKill – requires to stripe a data block in different channels, ranks, and banks; this kind of address mapping requires multiple banks opening to fetch one single data block, which is not energy efficient. Citadel enables the memory system to store a data block in one bank while protecting against a large granularity failure. To that end, this technique swaps the TSVs when one of them is faulty; furthermore, it provides 3-dimensional parity in three spatial dimensions to handle internal DRAM die failures. Finally, this technique implements dynamic sparing for faulty cache lines; when a 3-dimensional parity code restores a faulty data block, its row is remapped to a new spare location.

Similar to Citadel, RATT-ECC [57] employs a two-tiered error-correction technique to provide reliability for a 3D stacked memory. Using Reed-Solomon code in the first tier, RATT-ECC can detect large granularity failures and correct small granularity failures – such as TSV failures or single-bit errors. As a Tier-2 code, this technique also has a 2D parity code across banks and channels to correct errors detected by the first tier code. Unlike Citadel, RATT-ECC increases the refresh interval, thus introducing some additional errors, which can be corrected by its low latency tier-1 code. Hence, this method reduces the refresh energy as well.

In a similar way to Citadel and RATT-ECC, Jeon et al. [90] exploit a two-tiered error-correction method to protect 3D stacked memory from various types of failures. This technique uses SSC-DSD codes, CRC-8 technique, as its tier-1 code, which can correct

single-bit, column, and TSV errors; this technique employs a RAID5-like parity coding across channels to correct the multibit errors, which are not covered by the first tier error code. Similar to Citadel and RATT-ECC, for every read request, this technique makes two memory accesses. However, for a write request and error correction, the bandwidth overhead of this technique is lower than two other methods.

None of these proposals mentioned above – Citadel, RATT-ECC, and [90] – can protect against die or channel failures. Moreover, all prior techniques assume that there is a single unified memory controller with all knowledge of the entire HBM capacity to compute and store parity bits. However, due to the point that every HBM channel has its own memory controller, this assumption is not close to reality.

To address these issues, Jenga [117] proposes a solution to reduce the memory bandwidth overhead at a higher capacity overhead. In all prior methods, the second tier code is the Xor of multiple data blocks, which, along with those data blocks, is stored across different channels. This strategy increases the bandwidth overhead of error correction and write requests significantly. Jenga mitigates this issue by adding redundancy at the finer granularity; this reduces the bandwidth overhead at a higher capacity cost. Jenga splits a 72 Byte data block into two 36 Byte halves, computes their bitwise Xor, and spreads these three 36Byte subblocks of data in different channels and different dies. Therefore, Jenga achieves a lower bandwidth overhead with an affordable additional capacity overhead compared with prior works.

Two-tiered error protection [116], [186], [203] separates the correction and detection mechanisms to improve energy and capacity of reliability techniques. LOT-ECC [186] uses parity and RAID-like approaches to provide chipkill protection at a lower cost. Virtualized ECC (VECC) [203] uses a tier-1 code to detect different types of failure, but it cannot correct them. Then in a rare case that an error gets detected by the tier-1 code, VECC employs the second tier code to correct it. Since the tier-1 error code is required for every memory access, it is stored along with data in the same rank, in the ECC chip, i.e., the $9^{th}$ chip in the ECC DIMM. This technique saves the tier-2 error code separately in a page allocated by the operating system. Although implementing VECC on an ECC DIMM boosts its performance, it is not required.

Like VECC, Odd-ECC [116] exploits a two-level error code mechanism to define differ-

ent levels of protection for pages; these levels can be determined dynamically on demand for each page stored in the memory. Different levels of fault tolerance can be defined at the page granularity or the granularity of a region – e.g., standard regions such as the stack, heap, and global. Odd-ECC proposes a DRAM placement where ECCs are stored in separate pages, invisible to the application, and allocated by the OS.

This technique defines three levels of protection: "Tier zero (T0)", i.e., no protection, where no ECC page is allocated. The second level is "Tier one (T1)", where error detection and correction are supported for a single-bit error, and only error detection is provided for multibit errors; in this level, for every 56 pages, eight ECC pages are allocated to store tier-1 error codes. Finally, the full protected level is "Tier two (T2)" to correct multibit errors. In this level, every 49 data pages have seven pages containing the tier-2 code and eight pages allocated for the tier-1 code.

Kim et al. [97] present and evaluate a family of the single-tier error correction and detection codes for DRAM memory, called *Bamboo ECC*. Bamboo ECC can protect against various types of error on DRAM pins, providing up to the chipkill level. Bamboo ECC can correct more pin and chip errors compared with the current single-tier ECC approaches. By providing better detection capabilities, Bamboo ECC can reduce the silent data corruption (SDC) rate. This family delivers a fine-grained redundancy to achieve chipkill protection compared with prior techniques – 8b granularity compared with 8B in current chipkill techniques.

Bamboo ECC can improve memory bandwidth and storage overhead to protect all required types of failures – and chipkill protection – compared with prior techniques. One key advantage of Bamboo ECC is that it can provide the same protection level with fewer additional pins than previous techniques. For example, in a DIMM with 16 x4 DRAM chips, Bamboo ECC can correct one pin failure with two extra pins, while a SEC-DED code requires eight extra pins (2 DRAM x4 chips) to provide the same level of protection – 3.1% vs. 12.5% storage and pin overhead. However, in the case of single pin correction (SPC), Bamboo ECC is not compatible with off the shelf DRAM chips – they are either x4 or x8. This weakness is not applicable when this family provides the single-pin-correcting-and-triple-pin-detecting (SPC-TPD) or double-pin-correcting (DPC) techniques. To that end, one extra x4 DRAM chip is enough to provide four 8-bit redundant symbols, aug-

menting a DIMM with the SPC-TPD or DPC capability – one extra chip compared with two extra chips in the SEC-DED method. Bamboo ECC can also prepare the Quadruple-pin-correcting (QPC) capability or a chipkill protection with two additional x4 chips in a 16-chip DIMM.

While DRAM technology scales down to the smaller and more dense families, DRAM chips are becoming more error-prone. To address this issue, manufacturers have started augmenting DRAM chips with internal error correction codes, called *On-Die ECC*. XED [132] and DUO [77] try to exploit this on-chip redundancy to reduce the overhead of error protection techniques. XED makes the on-die ECC error correction available to the memory controller for error detection; using a parity-based error correction code, XED provides an efficient chipkill protection for an ECC-DIMM with nine x8 DRAM chips.

Contrary to XED, DUO tries to bypass the on-die ECC module and exploit the on-chip redundancy to strengthen rank-level ECC protection. DUO uses the Reed-Solomon code (RS) to protect a 64B data block – for example, for an ECC-DIMM with 18 x4-DRAM chips, DUO uses RS(72,64) to provide 12 check symbols for a data block. DUO exploits the on-chip redundancy to exert the Reed-Solomon code efficiently. Implementing a plain RS code cannot provide the single-device-data-correcting (SDDC). However, the key point is that if the failure device is spotted correctly, all its failures can be corrected using a burst erasure decoding technique [89]. Regarding this point, to spot the faulty chip, DUO does a brute force decoding search. In each trial of this search, DUO assumes that a different device is failed, applying the burst erasure decoding, and checking with RS code to spot the faulty chip. DUO proposes a new usage for the on-chip redundancy to mitigate its overfetching issue – this issue is due to the mismatch between the on-chip ECC codeword length and the memory bus width – thus reducing consumed energy.

## 3.3 Unified Integrity and Reliability

Due to the noticeable resemblance between security and reliability, proposing a solution with a combination of security and reliability metadata can mitigate the significant overall overhead of both effectively. Synergy [152] and IVEC [85] try to use integrity metadata to provide reliability, while Osiris [202] exploits the reliability metadata to provide integrity verification, and hence these techniques reduce the overall overhead of reliabil-

ity and security. In addition to Synergy (Subsection 5.2.3), IVEC [85] offers a combined solution for both integrity and chipkill. Unlike Synergy that exploits an ECC DIMM, IVEC supports chipkill for non-ECC DIMMs. IVEC borrows the idea of virtualized ECC (VECC) [203] to provide chipkill for non-ECC DIMMs; IVEC employs the MAC tag as a detection code, while the correction process is performed by parity bits.

Encryption counters in architectures using counter-mode encryption [149] must be correctly restored after a crash to allow the system to recover secure data stored in nonvolatile memories. The maintenance cost of these counters so that their persistence is guaranteed after the crash is significant. Osiris [202] exploits the error correction codes (ECCs) to accelerate the recovery process of these encryption counters. This technique claims that if plain-text data appended with its ECC is encrypted, ECC bits provide a sanity check for encryption counters. When a crash happens, first, Osiris recovers the last version of counters using ECC bits; then, it creates a Merkle tree and compares the computed root with the version that is stored in the CPU.

## 3.4 Smart Memories for Security

A memory system augmented with computational capabilities – called smart memory – provides security primitives more efficiently than a conventional one. In this section, we explore studies that implement different security features by adding computation components in the memory side.

Two works, InvisiMem [164] and ObfusMem [38], show how memory devices with logic capabilities can lower the overheads for both integrity verification and oblivious RAM. Since in DRAM stacked memories, such as HBM or HMC, DRAM dies are excluded from the TCB, the data still has to be appended with a hash tag and encrypted to protect against any data manipulations – such as Row-hammer attacks or cold boot attacks. However, providing data freshness does not require an integrity tree. Indeed, every data block does not need a separate counter. Instead, to guarantee the freshness of data, the processor and memory system maintain a global counter or timestamp. When writing a block, the processor generates a MAC tag for the encrypted data using the global counter and a hash algorithm, e.g., HMAC. In the memory side, the MAC tag will be regenerated, and if it matches, the data, its MAC, and the timestamp will be stored in the memory.

Obliviousness can be provided by smart memories very efficiently. The processor encrypts the data, address, and type of request to provide obliviousness; memory also encrypts data and sends it back to the processor in response to a read request to ensure that there are no similarities between a read and prior write requests. Note that read responses and write requests carry data, while read requests and write responses do not. Therefore, by revealing the length of requests/responses, attackers can infer the type of requests or responses. InvisiMem and Obfusmem equalize the shorter ones by adding a dummy data field to address this issue, which causes a trivial performance reduction.

Akin to ORAM, the mechanism mentioned above cannot hide the number of memory accesses and access time. Regarding the location, the access time may vary, which leaks applications' memory access patterns. To address this issue, InvisiMem dictates both memory and CPU to send packets at a constant rate. If there is no packet to send for a time slot, one dummy block will is sent. Note that conventional memory systems cannot send packets at a constant rate. In these memory systems, banks' states and several DRAM timing parameters dictated by the memory controller in the CPU side determine the response time.

Although exploiting a smart memory system is highly fascinating because of their low bandwidth overhead, there are multiple issues in this type of memory: first, we expand the TCB to embrace the memory logic layer. It is very challenging to include the memory system in the attestation process. Second, smart memories – e.g., HMCs or HBM with a logic layer – are still expensive with a limited capacity. Therefore, to replace all commodity DRAM chips with this type of memory to achieve the same memory capacity, we have to use a farm of smart memory packages connected in a grid network; this configuration leads to a tremendously high cost and area. Moreover, the connection between memory packages in the grid is not trusted, requiring some protection guarantees, which increases the overhead.

As an intermediate solution, Secure-DIMM [161] proposes a method to move some ORAM functionalities to the memory side to reduce the overhead of obliviousness significantly. Secure-DIMM is built upon commodity DIMM augmented with a secure buffer to do computation. In this technique, instead of inserting an ORAM controller in the CPU, which poses a significant bandwidth overhead on the memory bus, the ORAM con-

troller is outsourced to the DIMM. Therefore, the memory channel between DIMM and the processor does not observe any extra bandwidth pressure to provide ORAM primitives. The packets are encrypted by a global counter similar to what Invisimem and Obfusmem propose. Note that accessing different DIMMs and channels can reveal the memory access pattern; to eliminate this leakage, Secure-DIMM applies a traffic shaping strategy, which obfuscates the accesses to different DIMMs, channels, and ranks at a trivial performance cost. In Secure-DIMM, the overhead of integrity verification and freshness is similar to a conventional DIMM.

## 3.5 SGX Performance Enhancements

Since SGX imposes significant overheads on a secure system, a couple of studies attempt to alleviate these substantial overheads. In this section, we discuss some works trying to improve efficiency of an SGX-based system.

The transition between the EPC and non-EPC regions is the major overhead of SGX [36]. A significant number of studies reduce this dominant part of the SGX overhead [36], [140], [182], [183], [198]. SCONE [36] observes that in SGX, an enclave can access the non-EPC region with relatively low overhead compared with an EPC access. To leverage this observation, SCONE tries to run system calls outside the EPC. To that end, SCONE defines an asynchronous system call interface with shared memory to pass the system call arguments and return values. This shared memory also contains a variable to signal that arguments are ready, and the system call should execute. Another thread outside the EPC executes system calls, and thus the thread inside the enclave does not have to exit. Note SCONE itself is responsible for providing confidentiality and integrity protection for the data transferred from outside the EPC.

To further reduce the number of enclave transitions and decrease their overheads, SCONE also implements an internal threading process. Multiple application threads (say M) are assigned to several OS threads (say N). When a thread is waiting for a system call, another thread gets woken up and executes until the return values of the system call become available for the first thread. Like SCONE, Eleos [198] proposes a mechanism to handle page faults without exiting from enclaves, thus reducing the overhead of page faults in SGX. This technique allocates two regions, one in the EPC, called EPC++, and

another one in the non-EPC region, named as the backing store. Eleos maintains the page tables for these two regions. When a requested page does not reside in the EPC++, a page fault happens, but this page fault can be handled in the software level by moving the pages from the backing store to the EPC++; this page fault handler does not require enclave to exit.

Weisse et al. [198] break down the overall overhead of SGX into three parts: first, secure context switches, second, parameters and data transfer between application – which creates an enclave – and its enclave, and finally, memory access overhead – MEE cost to guarantee security primitives [82]. HotCall [198], like SCONE, provides an asynchronous system call interface for an SGX-based system to reduce the overhead of transition between enclave and non-enclave. To avoid context switching, HotCall uses an un-encrypted shared memory to communicate between a requester, the party that requests a call or an enclave, and a responder, the untrusted code which is waiting for a call request. When issuing a system call, the enclave copies the system call arguments, and the ID of the required system call into the shared memory and sets a signal to indicate that the requests are ready to execute. The responder continuously monitors the shared memory to execute requested system calls.

SGXKernel [183] implements asynchronous system calls at the user level, while prior work, e.g., SCONE, needs to load a special kernel to do so. In contrast to Graphene-SGX [185] conclusion and how Haven [44] deals with a Windows-based library OS, Tian et al. [183] observe that for library OSes running inside an enclave two requirements should be met; first, it is necessary to reduce the number of enclave transitions – a transition happens when an enclave calls an untrusted function outside the enclave. Second, we aim to have a small library OS to fit it into the EPC region. SGXKernel proposes a switchless architecture with two halves: one secure half residing in the enclave and the second one located outside the enclave. These two halves asynchronously communicate through shared memory. Moreover, similar to SCONE, this technique exploits an in-enclave multi-threading technique.

Prior works [36], [140], [183], [198] implement an asynchronous system call in which one sender thread sends a request through an untrusted shared buffer to a server thread, which executes system calls asynchronously. Although this solution seems entirely prac-

tical, Tian et al. [182] argue that it is not always a wise choice to allocate a separate core to reduce the number of enclave transitions. Tian et al. evaluate the performance improvement of these switch-less techniques for different benchmarks and conclude that switch-less calls can improve performance only if Ecall/Ocall functions are short and frequent. In other words, switch-less calls can achieve a significant improvement just for heavy workloads, and its improvement shrinks rapidly when the workloads' intensity decreases. Tian et al. propose a mathematical model characterizing a switch-less call's performance improvement, and hence, they define an efficiency factor that indicates this technique's effectiveness.

## 3.6 Side-Channel Attacks in SGX

SGX threat model does not cover any types of information leakage sources; therefore, an SGX-based system is vulnerable to a vast range of side-channel and covert attacks. In this section, we explore different side-channel attacks implemented over an SGX-based system and discuss the proposed solutions to mitigate them.

Xu et al. [199] demonstrate that a malicious OS can observe the memory access pattern of an application running in an enclave by tracking its page fault addresses. This attack is called *controlled channel attack* or *pigeonhole attack* [168]. Xu et al. [199] perform this attack on a JPEG application running in an SGX-based system to obtain the encrypted images. The compromised OS forces a sensitive application to encounter a page fault after every memory access by swapping out the touched pages. Therefore, the OS can observe the application's page-level memory access pattern, revealing the program's flow.

Sanctum [63] proposes a hardware extension for SGX to mitigate this issue. The operating system in Sanctum allocates a block of physical addresses to each enclave and allows the enclave to maintain its page table. Therefore, the OS does not have the visibility inside the enclave's page table, which hides the enclave memory accesses. This approach cannot support "demand paging." Hence, memory is overcommitted to the enclaves. If Sanctum intends to support demand paging, It has to use an Oblivious Ram (ORAM) technique to obfuscate page accesses, which imposes a significant overhead on system performance.

Similar to Sanctum, Invisipage [32] is a hardware solution that supports both obliv-

iousness along with demand paging for an SGX-based system. In this technique, the page management process is handled by the collaboration between the operating system and the enclave. The operating system is still in charge of page allocation; however, the enclave manages its own page table. Since the untrusted operating system performs page allocation, the page level access pattern can be revealed to the OS, when pages move between the EPC and non-EPC regions. To eliminate this leak, Invisipage exploits an ORAM based technique, named OPAM, to obfuscate the page accesses. This technique applies several optimizations to reduce the bandwidth overhead of the OPAM. To reduce the number of OPAM accesses, Invisipage defines a new region, called EPC-lite, where the integrity and confidentiality are supported at page granularity, like the non-EPC region. However, its page information is kept in the enclave page table, similar to the EPC region. The enclave has information of EPC-lite pages in its page table. Therefore, the enclave does not require to issue a page fault to access EPC-lite pages, and thus page transition between the EPC and EPC-lite is invisible to the OS and does not require obliviousness.

Shinde et al. [168] propose a compiler-based solution to eliminate leakage through page faults. This work tries to guarantee that applications' page-level access patterns cannot reveal the programs' execution flow. Shinde et al. propose an LLVM instrumentation technique to make page accesses utterly irrespective of the input secret's value. Therefore, the page-level access pattern does not contain any sensitive information. This proposal designs an efficient compiler to detect the sensitive branches in the application. Then, it makes the execution trees entirely balanced for these branches by adding some dummy execution blocks in the tree; this technique decouples the tree's fetch and execution steps. In the fetch step, all execution blocks will be fetched in every level of the tree, but only the required one will execute. A naive implementation of this technique imposes a considerable performance overhead (up to 4000×). The authors introduce some developer-assist optimizations to reduce the overhead, which requires a significant amount of manual effort by programmers.

Sinha et al. [169] implement an efficient compiler that provides page access obliviousness at a low overhead. The proposed compiler finds all of the secret-dependent conditional branches, and adds some dummy accesses to make read and write operations in both paths, if the branch is taken or not taken, completely identical. This technique also

provides a verifier to certify the program's obliviousness. The verifier checks whether or not the application's execution with different inputs can produce different page accesses. If it does so, the code does not satisfy the obliviousness requirements.

The verifier removes the compiler from the TCB, which makes the TCB significantly smaller than that in prior work [168]. Although both techniques, ( [168] and [169]), use the same approach to address the page fault attack, the former poses more bandwidth overhead than the latter. Moreover, the former's optimizations to reduce the bandwidth overhead rely on the developers, whereas the latter's can be applied automatically.

Some studies [58], [167] propose a solution for the page fault attack based on monitoring the OS and enclaves to recognize any suspicious activities from an untrusted OS or any unusual situation for enclaves to halt the system. T-SGX [167] employs Intel® Transactional Synchronization Extensions (TSX), which can implement a hardware transactional memory to protect the enclave against page fault attacks. T-SGX leverages the point that any exception occurring inside the TSX will not be directly delivered to the untrusted OS. Instead, the processor transfers control to the transaction's abort handler.

T-SGX partitions the code into multiple execution blocks, each of which is wrapped into a transaction, protected by a TSX. T-SGX places the code between transactions along with the transaction abort code on one page, which is called "Springboard." When an exception happens, control will transfer to the abort handler, and it determines whether the transaction should be restarted or terminates the enclave in case that the exception seems abnormal. The point is that only exceptions on Springboard are visible to the untrusted OS; the OS only can see the abort handler's address as a point where exception happens, and the real address where the exception occurred is hidden.

Exploiting Intel® Transactional Synchronization Extensions (TSX), similar to T-SGX, Deja Vu [58] empowers the enclave to detect that page fault attacks occur to terminate itself. Since SGX does not inform enclaves that a page fault, interrupt, or exception occurs, and enclaves cannot rely on the untrusted OS for this regard, an enclave cannot recognize that an AEX happens. Deja Vu enables enclaves to measure the execution time securely and accurately. As an exception or interrupt inside the enclave happens, the enclave has to exit, which increases its execution time significantly. Therefore, the execution time can serve as an indicator of an AEX's occurrence. Deja Vu instruments the program at compile time to

embed measured time for different blocks in the control-flow graph (CFG). Then, at the run time, by using a real-time clock, the enclave frequently measures the execution time of different paths in the CFG; if the measured time is suspiciously longer than before, it raises a flag for the enclave that OS may show a malicious activity, and Deja Vu terminates the enclave.

It is well-known that SGX does not have any protection against the side-channel or covert channel attack. Different levels of memory systems ranging from TLBs to the external main memory in an SGX-based system can leak sensitive information. In an SGX-enabled CPU, when HyperThreading (HT) is activated, one enclave thread and one non-enclave thread are likely to share a TLB, and they may use some similar entries. Therefore, the non-enclave thread can easily mount a cache attack against the enclave thread to obtain its access pattern. In SGX, entering and leaving the enclave flushes the corresponding entries in the TLB; therefore, an attacker can leverage this capability to flush entries in the TLB to perform cache attacks against another enclave – note that in this scenario, the attacker does not require HyperThreading to implement the attack.

Moreover, since Page Table Entries (PTEs) are also stored in the different cache hierarchy levels, they can also be exerted to implement cache attacks. PTEs contain different flags that can leak the enclave access pattern to a malicious OS without any page fault or even Asynchronous Enclave Exit (AEX). For example, when the page table walk translates a virtual address to a physical address in the page table entry, the accessed flag of the corresponding entry will be set to one, leaking to the OS which page table entry is recently accessed. The dirty flag in a PTE can also reveal the type of access, write, or read.

It is well-studied that SGX is vulnerable to cache attacks in all different levels of the cache hierarchy. Finally, SGX does not hide the DRAM memory access pattern, and since the DRAM memory controller, memory channels, ranks, banks might be shared among different applications, a DRAM timing channel attack is feasible [50], [190].

Exploiting these information leakage resources, Wang et al. [190] introduce a new memory-based attack, named Sneaky Page Monitoring (SPM), which does not need any page faults or AEX; hence it is more stealthy and more challenging to detect compared to the page fault attacks. Wang et al. implement three types of SPM attacks by monitoring the page table entries in conjunction with different methods to flush TLBs. Note that PTEs can be

exploited to implement an attack only if they are updated. To that end, the attacker needs to be able to flush the TLB entries. In the *basic PSM* or *B-SPM* attack, the malicious OS checks the PTE accessed flag to see whether it is set or not. Once it gets set to one, it means that the enclave touches the corresponding page. Then, the OS resets the accessed flag and waits for this flag to set to one again. To implement this attack effectively, the attacker needs to flush the TLB entries. The attacker needs to generate an interinterprocessor interrupt (IPI) for a different CPU core to shoot down the TLB.

The second SPM attack is *T-SPM* or time enhancement, in which, similar to B-SPM, the malicious OS focuses on the accessed flag of the PTE. Unlike B-SPM, in T-SPM, the attacker takes advantage of the timing leakage to reduce the number of TLB shootdowns. When in an attack, frequent TLB flushing is required, this strategy makes it more feasible. For example, for a secret-conditioned branch where only one path contains a loop, the attacker needs to flush the TLB once per loop trial during the loop execution to infer which path is taken by counting the number of accesses to the page – if the page is accessed multiple times, the path with the loop is taken. However, T-SPM suggests a more stealthy way of implementing this attack. If there is an entry page (say ENT) and exit page (say EXT) to enter and exit this piece of the code, the attacker can recognize the path by monitoring the PTEs corresponding to ENT and EXT and measuring the time interval between ENT and EXT access times. Hence, the number of TLB shootdowns will reduce dramatically because the attacker needs to flush only corresponding entries of EXE and ENT in the TLB once.

The third SPM attack is *HT-SPM* that does not require any interrupt to flush the TLB, and thus is more stealthy and more challenging. In this attack, the attacker takes advantage of HyperThreading to flush TLB entries. In Hyperthreading, threads running on the same physical core, share the TLB. The attacker can flush shared entries just by accessing some pages accordingly, without issuing any interrupts.

It is well-known that SGX is vulnerable to cache attacks [48], [78], [128], [159] and microarchitectural side channels based on speculation [56], [69], [126], [139], [187], [188]. SgxPectre [56] and SGXSpectre [139] mount a Spectre-like attack against an SGX-based system using the Branch Target Buffer. Since unauthorized accesses to the enclave memory does not trigger any page fault exception, a plain Meltdown attack cannot be performed

on an SGX-based machine. Foreshadow [187] tries to address this issue by clearing the "present bit" of the corresponding PTE, which forces SGX to issue a page fault exception if there is an access to the corresponding page. Then, similar to a Meltdown attack, the attacker can mount a Flush+Reload cache attack to obtain the secret. BranchScope [69] is another Spectre based attack that applies to all Intel® processors, including SGX. This attack poisons the shared direction branch predictor component of the branch predictor unit – contrary to prior works which use Branch Target Buffer (BTB) – to steer the victim to the wrong direction, forcing the victim to fetch the secret to obtain it through a covert channel – e.g., the shared cache hierarchy.

*Microarchitectural Data Sampling (MLS)* is a new set of vulnerabilities in Intel® processor – including the secure one, SGX, as well – that leaks secret data through CPU's internal buffers and registers. These weaknesses allow unauthorized speculative access to reach data stored in internal registers and buffers. Unlike other speculative execution attacks – e.g., Meltdown, Spectre, and Foreshadow – MLS attacks can obtain arbitrary in-flight data from internal buffers and registers, which has not yet been stored in the cache [121]. Using different buffers while HyperThreading is activated, researchers mount various MLS attacks against Intel® processors.

*RIDL (Rogue In-Flight Data Load)* [188] exploits speculative access to obtain arbitrary in-flight data from the Line Fill Buffers (LFBs[1]). The scenario to implement RIDL is straightforward; first, the victim loads or stores a secret, which is performed through some internal buffers such as LFBs. Then, when an attacker executes another load or store request, the CPU speculatively – without checking the virtual or physical addresses which are involved – fetches in-flight data from the LFB and stores it in the cache in the hope that this data is what the attacker requested. Finally, the CPU recognizes that the fetched data block does not belong to this request and wipes it from the cache; however, its remaining impact on the cache suffices for the attacker to obtain the secret of another application.

Similar to RIDL, *Fallout* [126], another MLS attack, exerts speculative access on store buffers to empower an attacker to gain a sample of other threads' data – it is called a "sample" of data because it may be a snapshot of data stored in an internal register or buffer

[1]It is called *Line Fill Buffer* in SandyBridge, while its name is *Miss Status Handling Registers (MSHRs)* in Xeon Phi.

which has not yet been stored in the cache. Intel® implemented a couple of optimizations in the store buffer to merge two store requests with the same addresses or consider a hit when a load request has the same address as prior store request, and the value of the store instruction will be returned for the load request. To accelerate this process, the processor predicts whether or not an address of a load request is the same as prior store requests. Based on this prediction, the processor speculatively returns the store's value for the load instruction and stores it in the cache. Later on, if the processor realizes that the previous prediction was wrong, the data will be wiped from the cache; however, its leakage can give enough information to the attacker.

Recently, Plundervolty [130] extracts the enclave cryptography key by injecting faults into an SGX-enabled CPU. To implement this attack, an attacker sets a fixed frequency for the CPU, while the working voltage can vary by writing on the MSR register – MSR with address `0x150` is to define the CPU voltage. Then the attacker mounts frequently the "Bellcore and Lenstra fault-injection attacks" [47] on an RSA function that uses the Chinese Remainder Theorem (CRT) optimization – Boneh et al. [47] show that the RSA cryptosystem is vulnerable to transient faults – to obtain the cryptography key.

# CHAPTER 4

# VAULT: A LOW OVERHEAD TRUSTED EXECUTION ENVIRONMENT

## 4.1 Introduction

A number of critical applications, e.g., electronic health records [93], 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 [62], [82]) 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 nonsensitive 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. Nonsensitive pages in the non-EPC region are accessed without security overheads.

On modern hardware, the overheads imposed by SGX are very significant. A simulation-based analysis is shown in the left third of Figure 4.1, and has been corroborated on the right by measurements on real SGX hardware, reported by other papers [36] [140]. 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 nonsecure 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 subbar). 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 subbar 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.

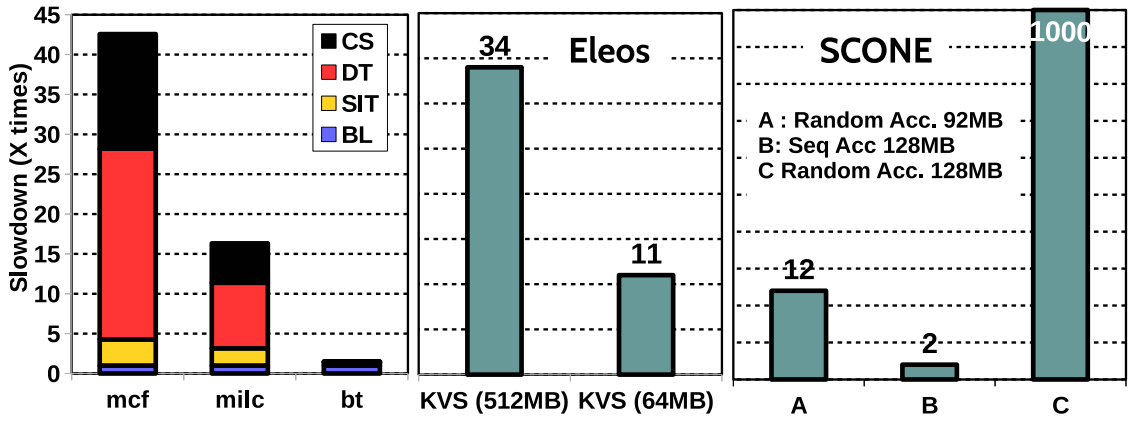



**Figure 4.1**: SGX overhead. Left-side: Slowdown for three different benchmarks with various numbers of page faults. The overhead is broken down in three portions, CS (Context Switch), DT (Data Transfer), and SIT (SGX Integrity Tree). The slowdown is against a nonsecure baseline system (BL). Middle: The slowdown of SGX in a real system for a Key Value Store with two different working set sizes [140]. Right-side: Slowdown for SGX in a real system for synthetic benchmarks, with random and sequential accesses, to different sizes of memory [36].

In brief, there is a large gap between EPC hit and miss latencies – 200 cycles vs. 40 K cycles [36]. A recent software solution, Eleos [140], 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 nonsensitive 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 nonsensitive 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

[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.

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, nonsensitive pages can disable subsets of their CIA operations and not be penalized. Even if a large fraction of pages are nonsensitive, the integrity tree overheads for sensitive 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 nonsecure 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%. Nonsensitive pages can be accessed without any bandwidth overheads, and sensitive pages are allowed to have a working set as large as physical memory.

## 4.2 Background

### 4.2.1 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 ap-

plications 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 [88].

- **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.

### 4.2.2 Merkle Trees

In a Merkle Tree (MT), the MACs of all the data blocks represent the leaf nodes. Each nonleaf 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 nonsecure 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 [107]. 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 nonsecure 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.

### 4.2.3 Bonsai Merkle Trees

- **Tamper-Proof Counters to Prevent Replay.** To alleviate the high overhead of Merkle Trees, Rogers et al. [149] 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 reencrypted 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.

### 4.2.4 Intel® SGX Baseline

- **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 nonsensitive 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 [62].

  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 nonsensitive 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 nonsensitve and disallows allocating them to any EPC pages.

- **Paging Overheads [140]**. 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 reentering 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 4.2, that we dub *SIT*. Every 512-bit node of 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 [197]). 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.

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

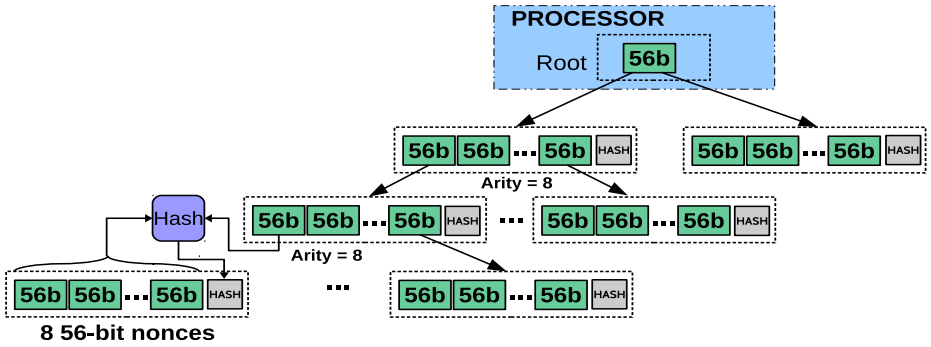



Figure 4.2: SGX integrity tree (SIT).

## 4.3 Proposed Techniques

### 4.3.1 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 nonsensitive. This subsection 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 subsection 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 metadata. As described in Section 4.2.4, 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 nonsecure 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 nonsecure 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 nonsensitive page, the typical encryption and integrity checks can be elided and nonsensitive page accesses are as fast as those in a nonsecure 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 4.2.4, 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 4.3.2), 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.

### 4.3.2 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 4.3). 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

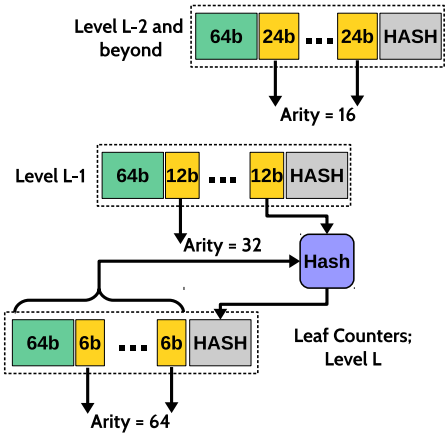



**Figure 4.3**: Variable Arity Unified Tree (VAUT).

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 4.3 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 reencrypt 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 nonleaf 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 4.3, 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

[3] A reset in a nonleaf node requires an update of the hash in all its child nodes. A reset in a leaf node requires a reencryption of all corresponding data blocks.

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 subsection, 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.

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 4.3.5), 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 4.4, 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

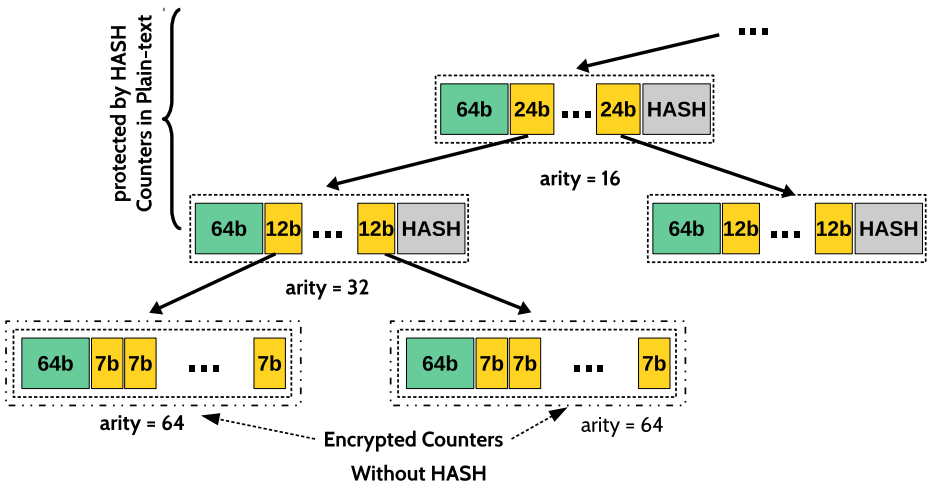



**Figure 4.4**: VAUT with encrypted Leaves (VAULT).

now on the critical path of the MAC confirmation, adding 40-80 cycles [85] 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.

### 4.3.3 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 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 [107].

But memory capacity is also an important metric that must be improved. As described earlier, the MACs introduce nontrivial 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 4.5, 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 4.5, compressible 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 [143]), 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

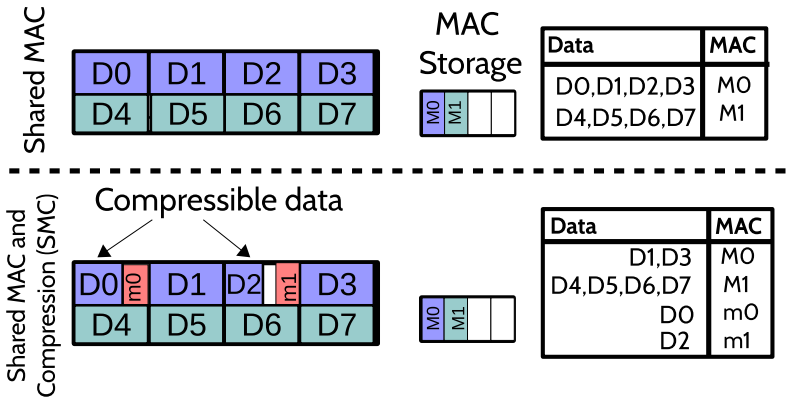



**Figure 4.5**: Shared MAC with Compression (SMC).

33mW [163]. The compression and decompression are performed entirely in hardware and are transparent to the operating system.

Since encryption increases data entropy, compression over a ciphertext block is not effective. Therefore, we first compress then encrypt data blocks. To encrypt a data block, we use the counter mode based encryption along with the AES128 algorithm; we split a 512 bit-data block into four 128-bit chunks ($D_0, D_1, D_2, D_3$). A ciphertext block is generated as $C_i = D_i \oplus PAD, for\ i = \{0, 1, 2, 3\}$, where PAD is the output of the AES algorithm. Using the counter mode encryption, the space that ciphertext and plaintext clocks occupy is the same; in other words, the counter mode encryption does not change the space generated by compression to carry the corresponding MAC.

In a write request, a data block is first compressed, then encrypted; finally, the corresponding MAC – and the ECC block to provide reliability – is generated upon the ciphertext. In a read request, the received MAC and the corresponding ECC are verified; the data block is then decrypted and decompressed. We use the "encrypt-then-MAC" method to protect data from vulnerabilities such as padding oracle attacks that might be generated by using the "MAC-then-encrypt" method.

### 4.3.4 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 nonsensitive data.

Due to MAC allocation at page granularity in the eviction tree (Section 4.2.4), 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.

### 4.3.5 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 nonleaf 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 subblocks $L_0$, $L_1$, $L_2$, and $L_3$. We create a new subblock $L_x = L_0 \oplus L_1 \oplus L_2 \oplus L_3$, where $\oplus$ represents XOR. The subblocks, $L_x$, $L_1$, $L_2$, $L_3$ are then encrypted to create 128-bit subblocks for the encrypted leaf node $L'$. Each subblock 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 subblock and the corresponding 76-bit counter stored in the parent of the leaf node. During decryption, the reverse operations are performed: subblocks $L'_x$, $L'_1$, $L'_2$, $L'_3$ 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 subblock $L'_*$, it results in a modified decrypted subblock $L_*$, and eventually a modified subblock $L_0$. Since the attacker does not know the CPU's private key, from the perspective of the attacker, subblock $L_0$ is

a random unknown subblock. 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 subblock $L_0$, 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 XORing the subblocks, we ensure that any modification to $L'$ results in a random global counter value.

### 4.3.6 Discussion

- **Capacity Overhead.** Table 4.1 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 4.2.4).

- **Compression and Encryption.** Compression and encryption are frequently used together, e.g., in file systems such as NTFS [125], ZFS [66], and Apple's HFS [115]. Compression does represent a side-channel – if system behavior can be observed, an attacker can estimate 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 [96]. If such leakage is deemed critical, compression could be performed at a coarse granularity, or with an element of randomness.

- **VAULT Implementation.** The modifications that are required to implement VAULT have to be applied at the hardware level. To implement VAULT, we have to modify the finite state machines in the MEE. A compression block is required to compress/decompress data blocks. The tree controller – which is responsible for main-

**Table 4.1**: Memory capacity overhead for different integrity techniques. Except for SGX (Baseline), other schemes use one unified tree for the entire 16GB memory space.

| 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%** |

taining the integrity tree – has to be modified. Two extra fields must be added to the EPCM entries: one field containing a pointer to the page MAC values to support the ODMA method. Second field – named SENSITIVE – which shows whether the corresponding page is sensitive. This field enables VAULT to allocate sensitive pages across the physical memory. Note VAULT assigns one EPCM entry to every physical page. Like TLB entries, EPCM entries should also be cached in the cache hierarchy to ensure that the corresponding EPCM entry is available on-chip for a TLB hit.

## 4.4 Methodology

To evaluate our techniques, we conduct cycle-accurate simulations with 21 workloads from two benchmark suites: SPEC2k6 [84] and NPB [41]. These benchmarks are described in Table 4.2, along with their working set size, the number of page faults per 50 million instructions, and compressibility with the Base-Delta-Immediate [143] 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 [17]; 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 [54]. Table 4.3 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

**Table 4.2**: Benchmark's specifications. Comp (Compressibility in percentage), WS (Working Set size in MB), and PF (average number of page faults in 50M instructions).

| **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 4.3: Simulator parameters.

| 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 [16], |
| 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 |

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 nonsecure 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 [113]. We consider 96 MB memory for the EPC with the clock algorithm [51] 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.

## 4.5 Results

### 4.5.1 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 4.6 shows execution time for these four cases for each benchmark, for an 8-core model, normalized against a nonsecure baseline. In all cases, VAULT incurs a lower execution time overhead, proportional to the memory bandwidth overhead. As shown in Figure 4.6, 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.

The average breakdown of memory traffic for MT, BMT, SIT, and VAULT is shown in Figure 4.7. 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.

### 4.5.2 Evaluation of Reset Overhead and VAULT

As we explained in Section 4.3, 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 4.8 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

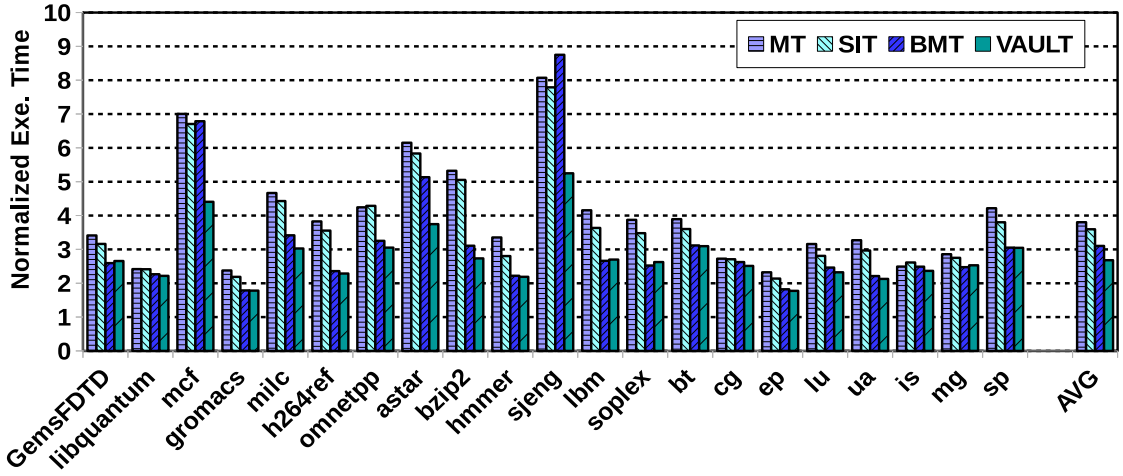



Figure 4.6: Execution time for MT, BMT, SIT, and VAULT, normalized against a nonsecure 8-core baseline. The trees cover the entire 16GB memory space.

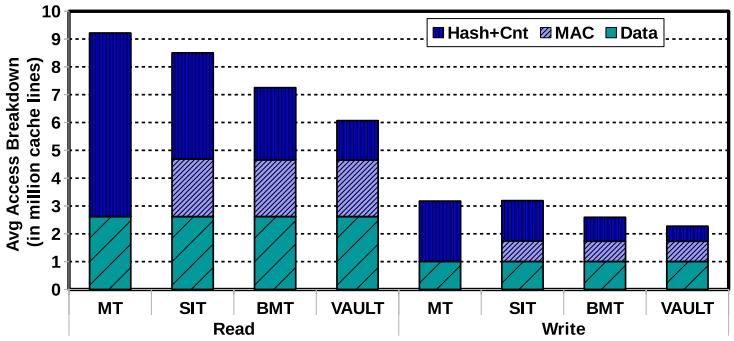



**Figure 4.7**: Average access breakdown for reads and writes in MT, BMT, SIT, and VAULT.

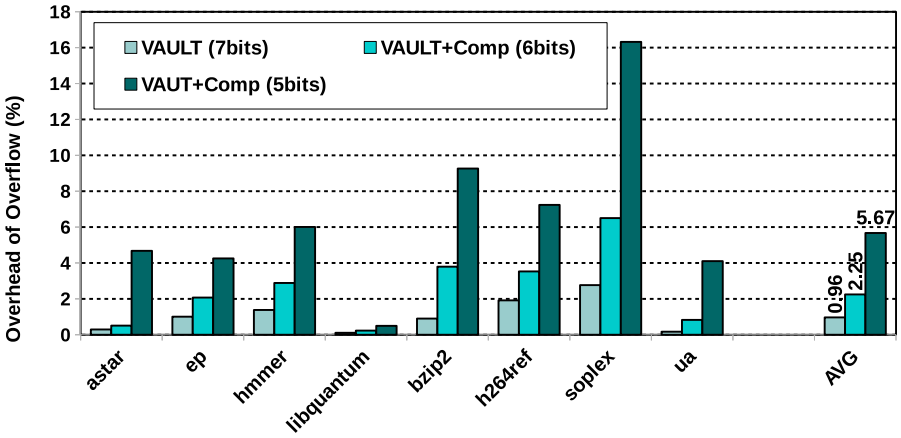



**Figure 4.8**: Execution time overhead introduced by counter reset handling. This graph only shows the 8 most affected benchmarks.

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 [85], we see nearly zero impact in the 8-core bandwidth-constrained model.

### 4.5.3 Evaluation of SMC

Figure 4.9 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 4.1. 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

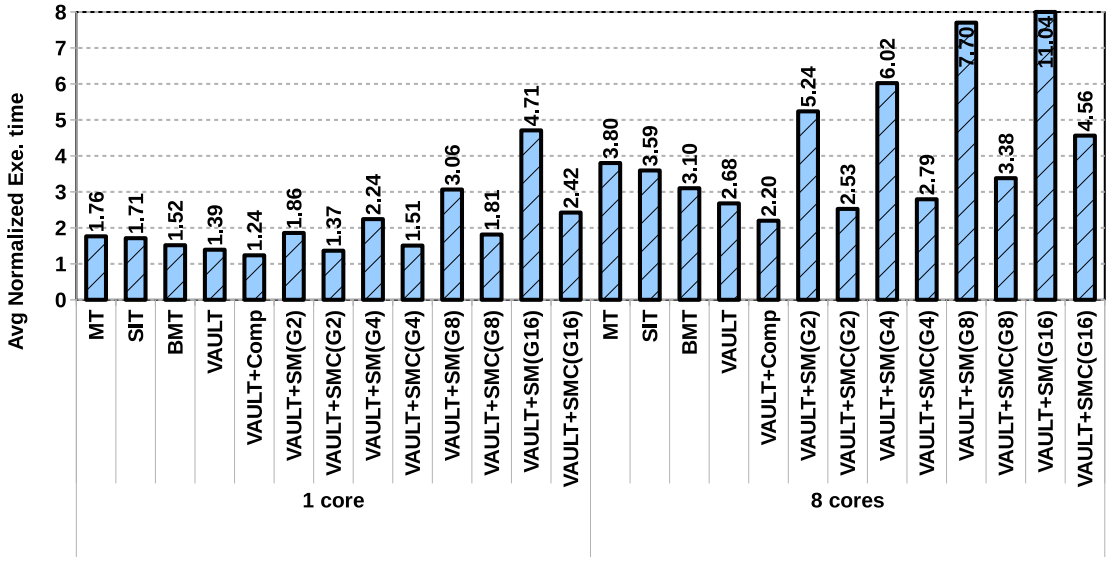



**Figure 4.9**: Average normalized execution time after applying the SMC technique with different group sizes, for varying core counts.

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 nonsecure 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.

### 4.5.4 Impact of Caching the Integrity Tree Nodes

We evaluate the impact of growing the sizes of the hash cache in Figure 4.10. We see that the hash cache shows steady improvements in going from 8 KB to 32 KB to 128 KB. The improvement increases as the number of enclaves increases.

### 4.5.5 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 [140], 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

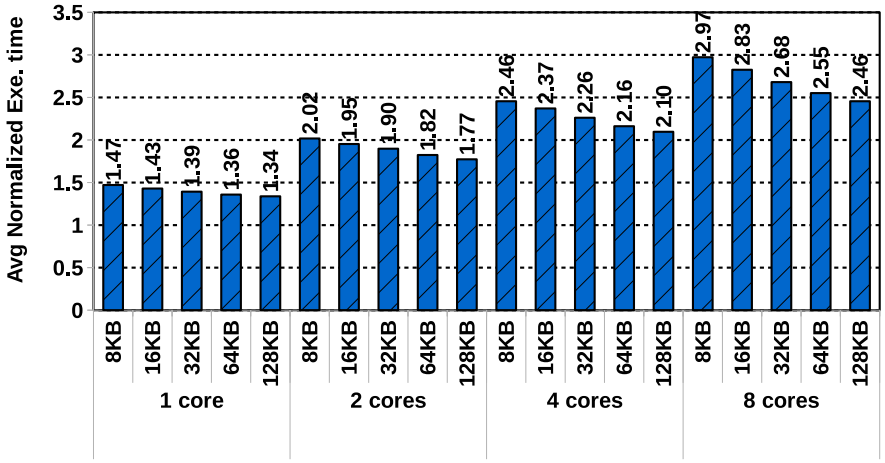



**Figure 4.10**: Normalized execution time as the size of hash cache changes from 8 KB to 128 KB per core.

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 [140]. Note that, as mentioned in Table 4.1, SGX and Eleos have negligible capacity overhead. Therefore, we also consider VAULT with SMC(G4), to make a fair comparison.

Figure 4.11 shows the slowdown of SGX, Eleos, VAULT, and VAULT+SMC4 for a single-core model, with respect to a nonsecure system. Eleos outperforms SGX by 2.3× and VAULT+SMC4 outperforms Eleos by 1.61×. When the number of enclaves increases

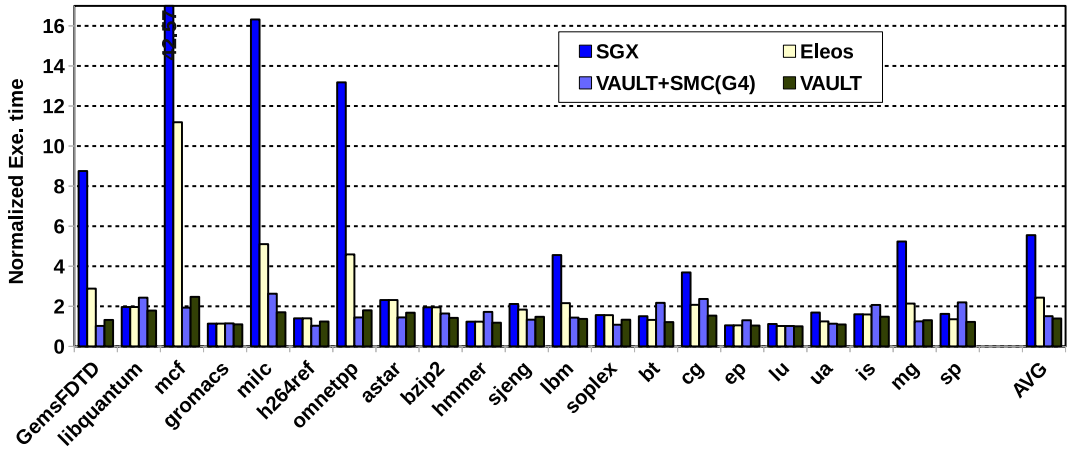



**Figure 4.11**: Execution time for SGX, Eleos, VAULT, and VAULT+SMC4, normalized against a nonsecure 1-enclave system.

(Figure 4.12), 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.

### 4.5.6 Summary of the Proposed Methods

In this section, we summarize the impact of each proposed method on the overheads of data integrity verification. Figure 4.13 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.

## 4.6 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 nonsensitive pages. However, naively growing

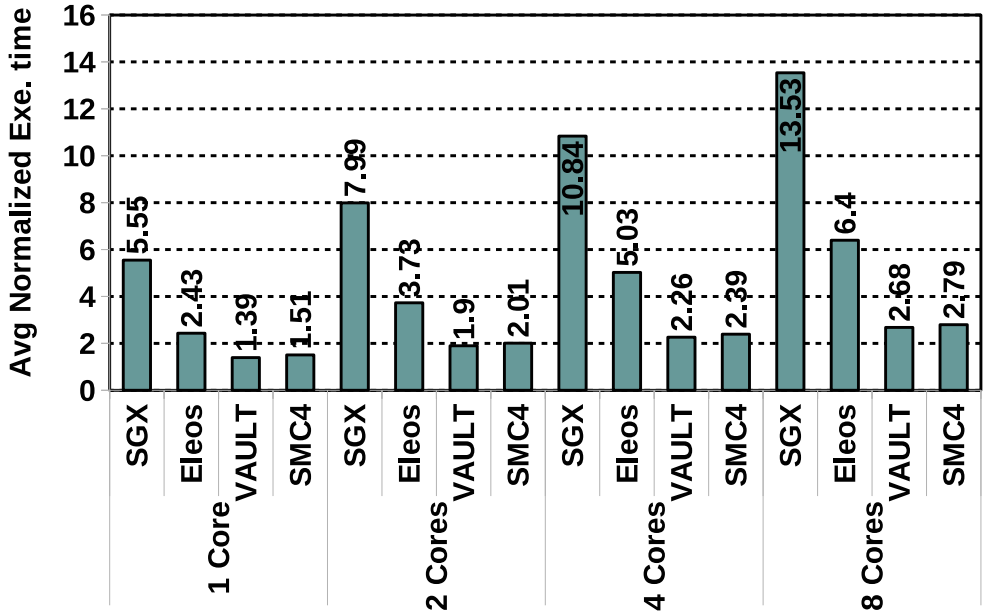



**Figure 4.12**: Average normalized execution time for SGX, Eleos, VAULT, and VAULT+SMC4 (shown by SMC4) when the number of enclaves varies.

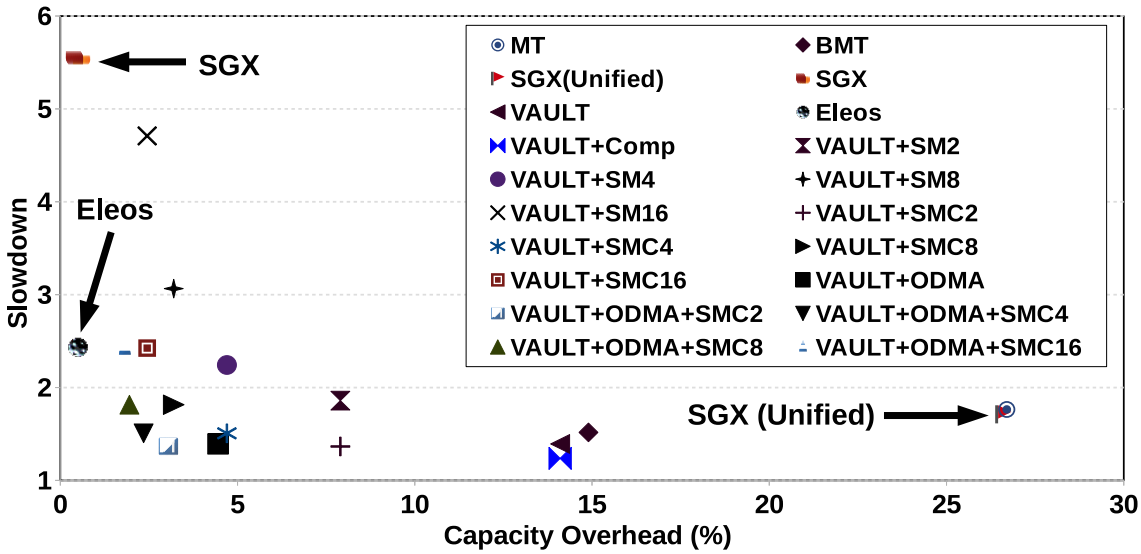



**Figure 4.13**: Comparison of different proposed methods.

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.

# CHAPTER 5

# ITESP: COMPACT LEAKAGE-FREE SUPPORT FOR INTEGRITY AND RELIABILITY

## 5.1 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 coscheduled 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 coscheduled thread can modify the victim's memory space with a row hammer attack [101], [160]. 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 [148]. 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) [62], [82], [120].

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× [36], [140], [181]. 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 [82], [149] 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 [106], [107], [152], 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 [181], Morphable Counters [151], and Synergy [152]. 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 interapplication 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 5.3.2 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 over-

head 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.

## 5.2 Background

### 5.2.1 Threat Model

We assume a threat model and security guarantee similar to popular memory security techniques [62], [82], [120]. 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 coscheduled 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 [82] 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. coscheduled applications on a processor share a number of resources; such sharing introduces side channels and vulnerabilities, e.g., in data/instruction caches [99], [193], branch predictors [69], coherence directories [200]. As we show later, the shared integrity tree and metadata cache can also be exploited by a coscheduled 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 [145], [194], [200].

### 5.2.2 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 [82], 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 [181] 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 [149]). This further improves the arity of the tree and shrinks its depth. Morphable Counters [151] 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 [106]. 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.

### 5.2.3 Synergy

In the Synergy proposal, Saileshwar et al. [152] 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 5.1 shows how data and metadata are organized for a baseline memory system and for Synergy [152]. 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.

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 5.1. The first parity bit captures the parity of

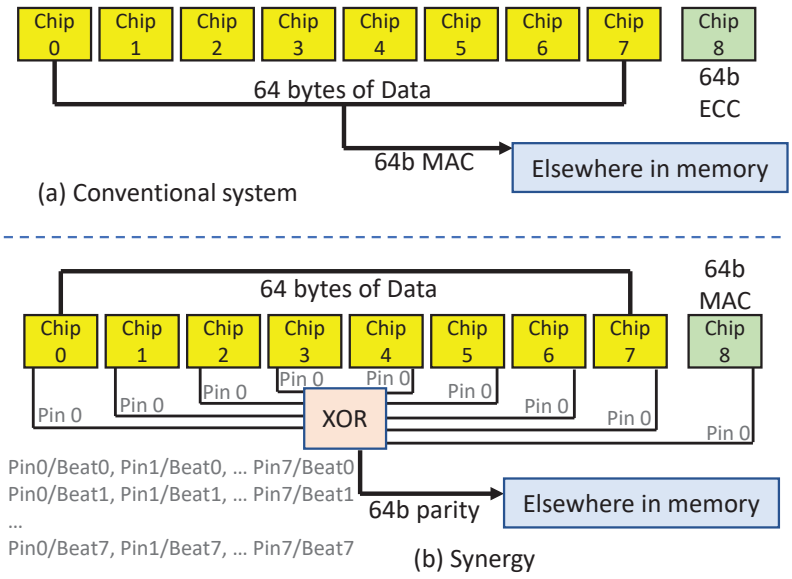



**Figure 5.1**: Data organization in baseline memory and Synergy.

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 [171], [172].

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 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.

To provide chipkill support for the integrity tree nodes, Synergy stores the corresponding parity blocks in the ECC chip along with the tree node blocks in the same memory rank. Therefore, for read and write requests, only one memory access is required to fetch one block of tree nodes along with its parity block. For error detection and integrity verification, MAC values embedded inside the tree node blocks are employed; the corresponding parity blocks – which are fetched along with tree node blocks – are used for error correction. Hence, whether or not there is any mismatch in the MAC values, to provide integrity verification and chipkill support for tree node blocks no extra memory access is required.

The Synergy approach relies on write masking so that only the parity bits of a modified block can be updated. Some systems [19], [29] 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 [124]. Thus, while Synergy works correctly and effectively in

[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.

theory, it is not compatible with all systems. Our proposed solution does not require write masking.

### 5.2.4 Motivation

A key drawback in prior work is that they implement a single integrity tree for all physical pages, leading to interapplication 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 5.4. 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 5.2 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× lower in the Large multiprogrammed 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 multiprogrammed interference.

Figure 5.3 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 multiprogrammed 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

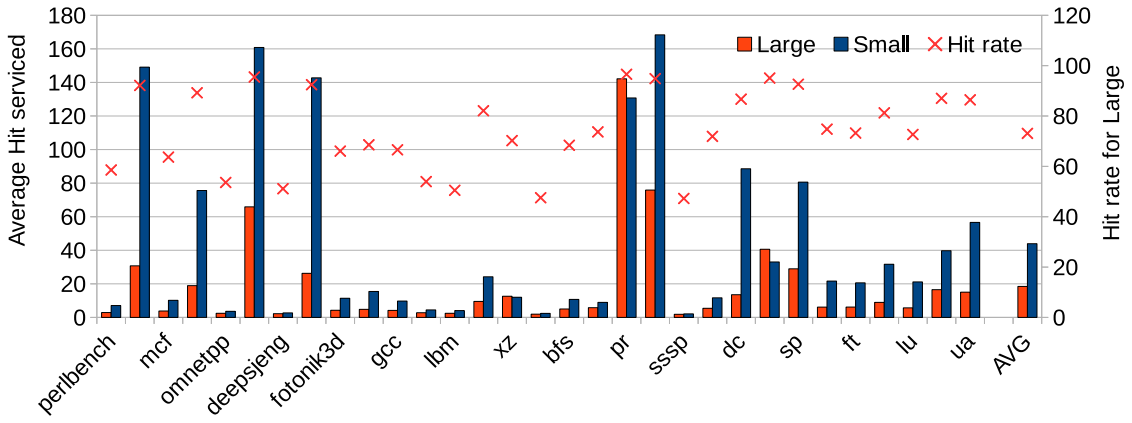



**Figure 5.2**: Metadata block utilization while in cache in VAULT (left Y-axis) and metadata cache hit rate (right Y-axis).

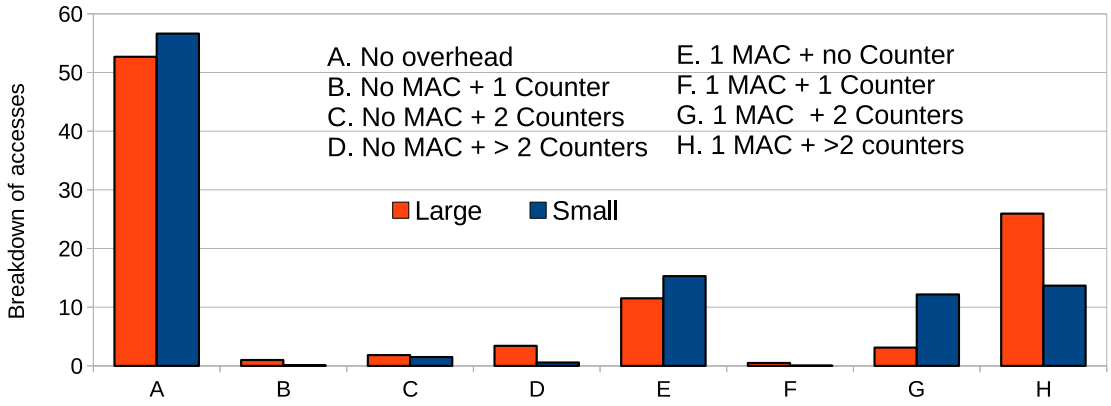



**Figure 5.3**: Breakdown of metadata access patterns.

analysis helps us understand the two main causes of metadata overheads. The take-home from Figure 5.2 is that interprogram interference greatly diminishes the utility of metadata cache entries. Figure 5.3 shows that because of multiple separate metadata structures, data block misses often require multiple metadata memory accesses.

## 5.3 Isolated Tree with Embedded Shared Parity

### 5.3.1 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 interprogram 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 reencrypted. 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 reencryption 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 than usual latency because it has to wait for reencryption 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 5.4 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 intermingled, 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

[3]Similar to prior attacks [86], 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.

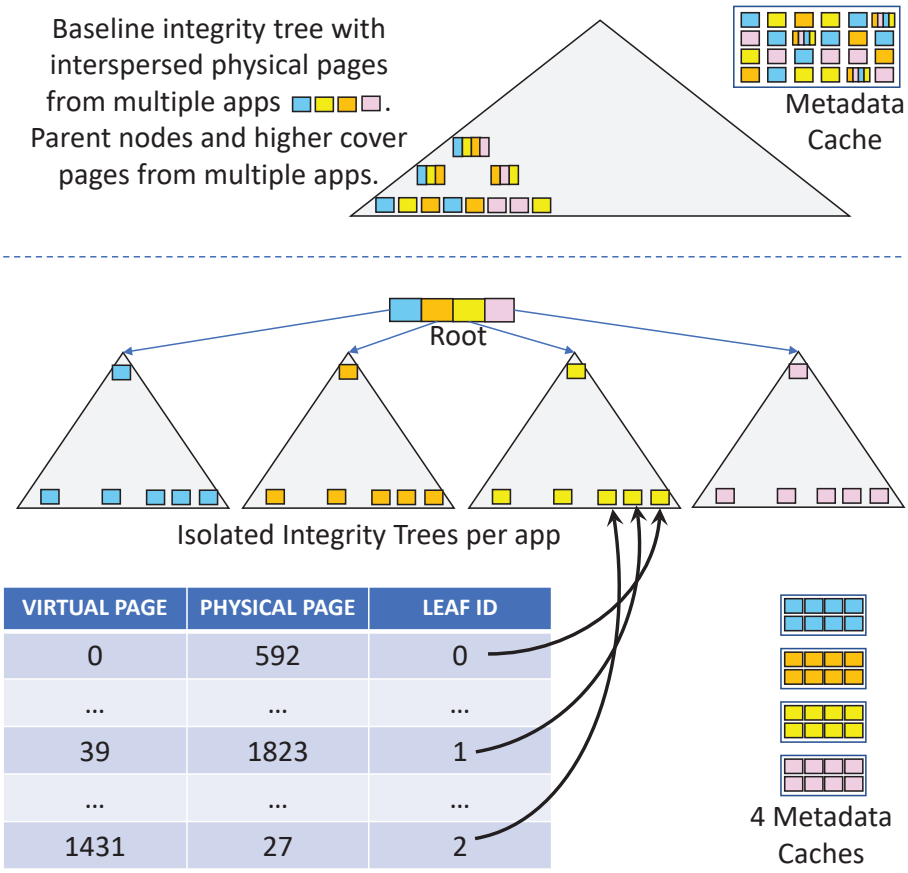



**Figure 5.4**: Isolated integrity tree. Baseline integrity tree and metadata cache for 4 apps (top). Isolated integrity trees and metadata caches (bottom).

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.

### 5.3.2 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 [82]) 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 5.5 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 5.5A, 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 5.5A shows the latencies experienced by the attacker when the victim is trans-

[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.

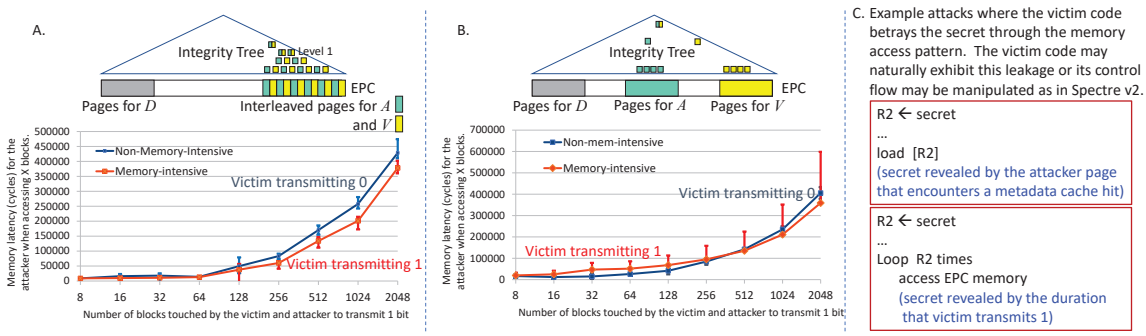



**Figure 5.5**: Covert channel demonstrated on an SGX v1 system. Observed latencies by the attacker and victim enclaves when pages are interleaved (A) or isolated (B). Two example victim code vulnerabilities (C) are also shown.

mitting 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. 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 5.5B 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 coscheduled VM. Alternatively, a victim program's memory intensity or memory access pattern can betray the secret (see examples in Figure 5.5C). Similar to the Spectre attacks [99], 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 5.5C 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 5.5C 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.

### 5.3.3 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 5.5, 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 5.3.7, even this impact can be mitigated.

- **Parity Read-Modify-Write.** Shared parity has one significant drawback. Without shar-

ing, 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.

### 5.3.4 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 5.6, 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 5.6 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 5.6, 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 down-

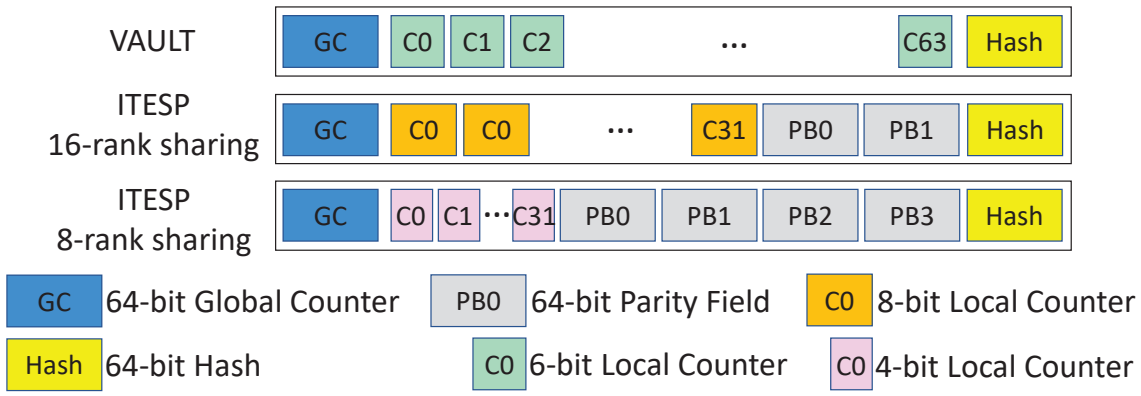



**Figure 5.6**: A block of counters in VAULT and ITESP.

side 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 5.7 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 5.5.

### 5.3.5 Implementation Details

- **Write Masking.** Many systems [19], [29], [124] 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.

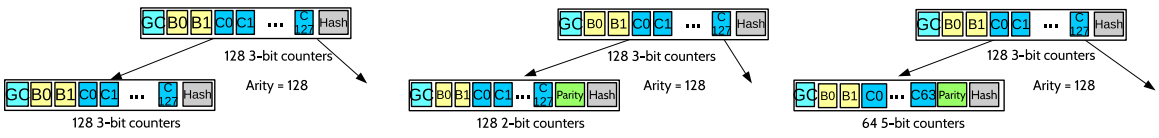



**Figure 5.7**: Different integrity trees with Morphable Counters: (a) SYN128: Arity of 128 throughout; (b) ITESP 64: Arity of 64 at leaf level and 128 at other levels; (c) ITESP 128: Arity of 128 throughout.

- **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 5.1 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.

- **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 5.5 and find an address mapping policy that balances these competing forces. In essence, four consecutive cache blocks are placed in a single

**Table 5.1**: Metadata memory capacity overheads.

| 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% |

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 [137] 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.

### 5.3.6 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 5.6 and 5.7, 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 [71], [162], [189], [191], [192].

### 5.3.7 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 reconstruction 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 [158] 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 quantify in Table 5.2.

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.

**Table 5.2**: Summary of SDC and DUE rates per billion hours for Synergy and ITESP.

| 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 |

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 [171].

- **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 [171]. 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 multidevice 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 $9 \times 2^{-64}$ (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 multidevice 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: multichip error and no matching MACs.** This is the common case during a multichip 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.

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 microsecond window, a chip-level failure will be detected within that window, triggering a correction and subsequent scrub. This would shrink the window for multierror incidence from an hour to a few seconds, thus lowering its probability by three orders of magnitude.

## 5.4 Methodology

We evaluate our techniques on 31 workloads, including 15 from SPEC2017 [49], 6 from GAP [45], and 10 from NAS [41]. We use Pin [113] 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 multiprogrammed 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 [54], a trace-based cycle-accurate memory simulator. Tables 5.3 and 5.4 show the specifications of our simulator and our benchmarks, respectively. We also identify the 15 most memory-intensive benchmarks in Table 5.4 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 [9] to estimate the power of each memory chip. System energy is estimated with similar assumptions as the Memory

**Table 5.3**: Simulator parameters.

| | |
|---|---|
| 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 [16], |
| 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, $t_{REFI}$ = 7.8$\mu s$, $t_{RFC}$ = 640 ns |

**Table 5.4**: Benchmark specifications. The 15 most memory-intensive benchmarks are shown in bold font.

| **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 |

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.

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 sensi-

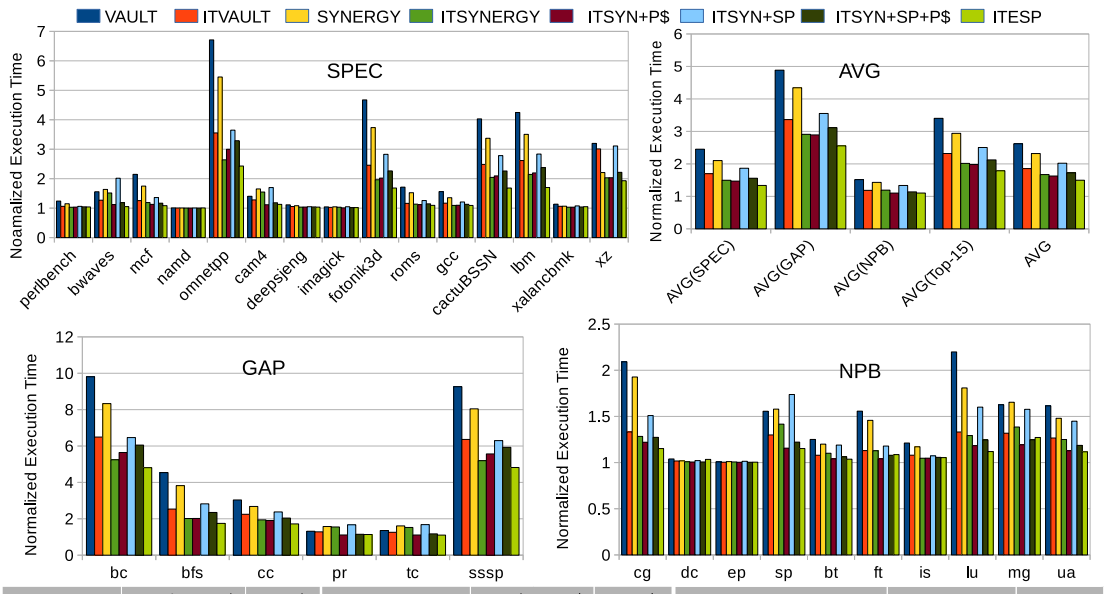
tivity analysis for the metadata cache size. Note that prior work has already shown that separate metadata caches work better than placing metadata in a larger LLC [152]. In the secure baseline (VAULT [181]), a 32 KB cache is used to store counters and integrity tree nodes, while a second 32 KB cache 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 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) [8] 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.

## 5.5 Results

In Section 5.5.1, we first examine performance and energy improvements for a baseline that integrates the Synergy and ITESP techniques into an integrity tree modeled after VAULT [181], 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 [151], i.e., a tree with even higher arity (64 and 128) and small local counters susceptible to high overflow rates.

### 5.5.1 ITESP for VAULT and Synergy Baselines

Figure 5.8 shows the execution time for the most relevant systems, all normalized against a nonsecure 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 5.4. 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,




| | 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 |

**Figure 5.8**: Execution time for the secure VAULT baseline, Vault with isolated trees and metadata caches (ITVAULT), VAULT+Synergy baseline (SYNERGY), VAULT and 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 nonsecure baseline. Assumes 4 cores and 1 memory channel. The benchmarks are organized by the suite.

Figure 5.9 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 5.8), with 20% lower metadata overhead (Figure 5.9). Note that the parity write traffic in baseline Synergy is high because it isn't cached.

It is worth mentioning that the average execution time for the VAULT technique – shown in Figure 5.8 – is slightly higher than what we have reported in Subsection 4.5. The reason is that the applications examined in the ITESP experiments are more memory-intensive compared with the ones used in Subsection 4.5, which leads to higher overhead for security features. The differences become wider when the ITESP results for the 15 most memory-intensive benchmarks are considered.

Adding isolation to both VAULT and Synergy has a significant performance impact, yielding performance improvements of 46% and 45%, respectively. This is primarily be-

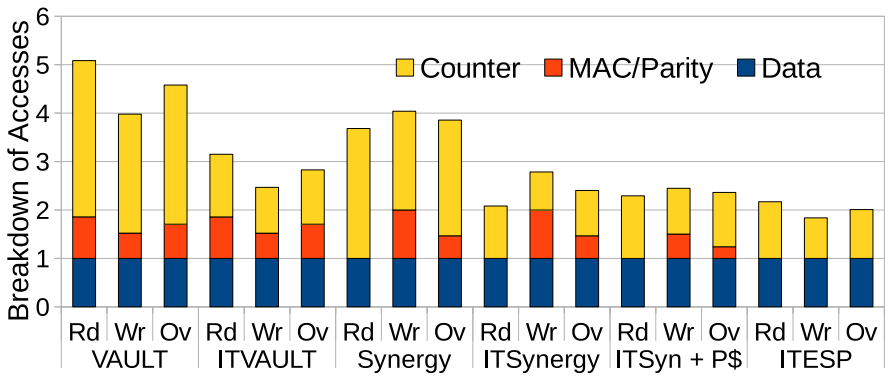



**Figure 5.9**: Breakdown of data+metadata accesses for each read and write operation. Averages are reported for the top-15 memory-intensive benchmarks.

cause 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 5.9) by avoiding interprogram 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 5.8 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.

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 5.9, 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 5.10 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.

### 5.5.2 Sensitivity Analysis

- **Core Count.** To understand the impact of executing a larger number of applications, Figure 5.11 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 interprogram 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.

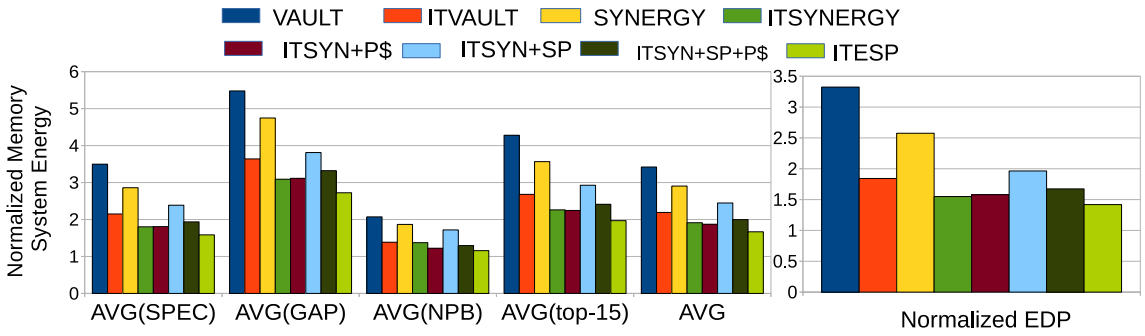



**Figure 5.10**: Normalized memory energy (on the left) and normalized average system energy delay product (EDP, on the right) for the same models described in Figure 5.8.

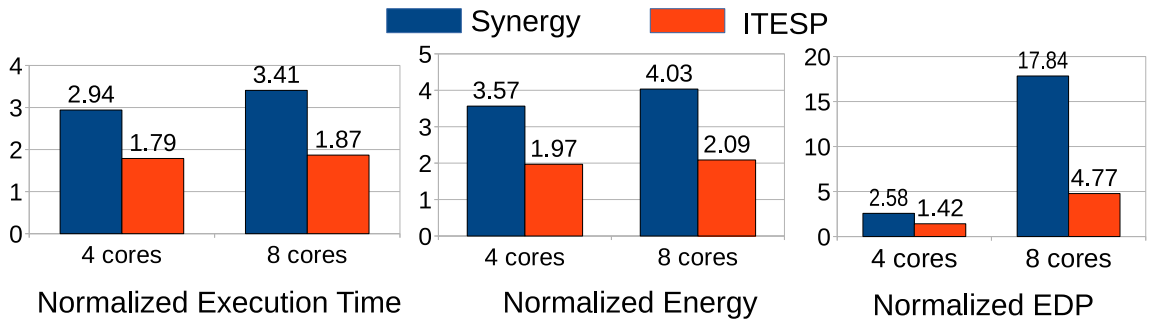



**Figure 5.11**: Execution time, memory energy, and system EDP for a 4-core model with 1 channel and an 8-core model with 2 channels, normalized against a nonsecure baseline.

- **Size of Metadata cache.** Figure 5.12 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 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.

### 5.5.3 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

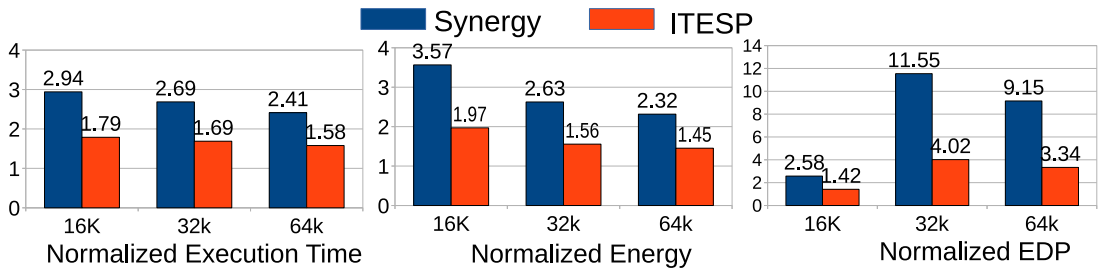



**Figure 5.12**: Execution time, memory energy, and system EDP for various metadata cache sizes, normalized against a nonsecure baseline. The bars represent averages over top-15 memory-intensive benchmarks.

that can balance the two. Figure 5.13 summarizes 4 relevant address mapping policies. Figure 5.14 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.

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 suboptimal 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.

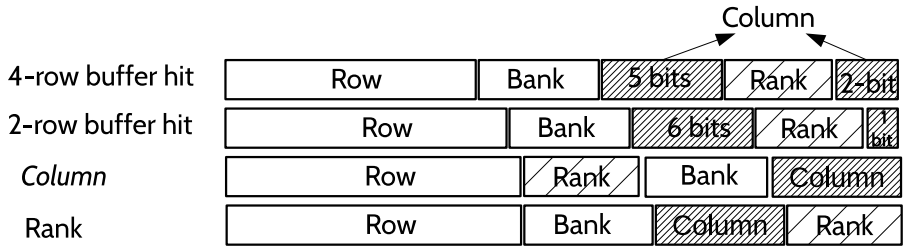



**Figure 5.13**: Address mapping policies for a 1-channel config.

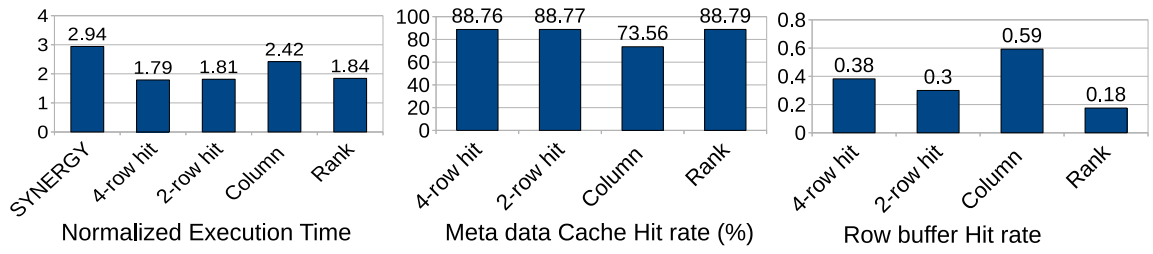



**Figure 5.14**: Impact of address mapping policies on performance, metadata cache miss rate, and row buffer hit rate (assuming 4 cores and 1 channel).

### 5.5.4 ITESP with Morphable Counter Baseline

Finally, we show that ITESP is also compatible with innovations [151] that exploit counter value locality to increase tree arity. Recall the three configurations described in Figure 5.7, 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 5.15 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).

## 5.6 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

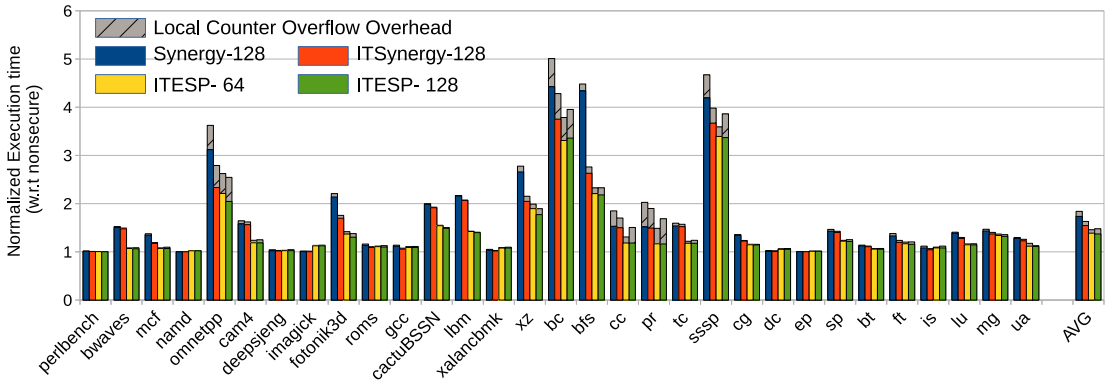



**Figure 5.15**: Normalized execution time (incl. local counter overflows) for Synergy and Morphable Counters (Synergy128), Synergy128 with Isolation, and ITESP with Morphable Counters (ITESP 64 and ITESP 128). Assumes 8 cores with 2 channels.

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.

# CHAPTER 6

# CONCLUSION

This dissertation introduces several new structures along with multiple techniques to alleviate the overhead of metadata maintenance in a trusted environment. We state that there are two major problems with the current implementation of integrity verification techniques, which significantly impact their efficiency and security. First, they maintain a couple of large metadata structures, which imposes substantial bandwidth and storage overheads on the system. Second, these metadata infrastructures leak information. In this thesis, we addressed both issues.

We propose a shorter and more cacheable integrity tree to improve its memory bandwidth overhead. We also introduce a more compact structure to store MACs, which reduces its storage overhead significantly, and as a result, improves its bandwidth overhead as well. We notice that significant metadata overhead forces a trusted environment to curb the size of the secure memory region – as the PRM size in Intel® SGX is 128 MB. This size restriction leads to other problems; if the sensitive information of an application does not fit into the given secure memory, the secure pages have to be swapped between the secure and nonsecure memory regions. This process has a significant performance overhead because it requires an OS system call, context switching, page copying, security checks, and metadata updating. In Chapter 4, we propose VAULT, a low overhead TEE that overcomes this issue. VAULT provides sensitive applications with a scalable, low-overhead, secure memory, where the overhead does not grow dramatically as the applications' working set size increases.

We notice that Synergy, the state-of-the-art technique that has a combination of error-correction and integrity metadata to provide both reliability and security at lower overhead, still suffers from maintaining two relatively large metadata structures. We take one step forward in Chapter 5 to further improve the metadata overhead by sharing a

reliability metadata block across multiple data blocks to lower its footprint. This technique enables us to embed the reliability metadata into the integrity tree. Therefore, we propose a metadata structure, which provides all required metadata to support reliability and integrity at lower bandwidth and storage overheads relative to Synergy.

Moreover, we isolate applications by separating their metadata structures, which not only improves performance due to the higher cacheability, but also eliminates potential information leakage. Altogether, we proposed ITESP, a leakage-free, unified, and low-overhead metadata structure to provide integrity and reliability.

Before diving into our contributions in this dissertation, we briefly explain multiple basic concepts in the security field, including "Trusted Execution Environments (TEE)" in Chapter 2. Since in different parts of this thesis, Intel® SGX is chosen as the baseline to compare with our proposals, we also elaborate on various aspects of this commercial TEE. Furthermore, we take a look at other proposals in the literature, which try to reduce the overhead of reliability, security, and security along with reliability support (Chapter 3).

## 6.1 Future Work

There are still multiple challenges to tackle and transform into opportunities to further improve our secure systems. In this section, we discuss some of these lines of research, which can be performed as follow-up studies to this thesis.

- **Emerging Memory Systems**. New DRAM families continue to be scaled down, which makes them more error-prone. To address this issue, the memory manufacturers place on-chip ECC codes and increase the number of error correction chips on the DIMM. Considering these new features, we can design a more efficient system to provide integrity verification and reliability.

  New High Bandwidth Memory (HBM), i.e., HBMv2, and HBMv2e, have become cheaper and denser with higher capacity and bandwidth, and as a result, more commonplace in computer systems, especially in current graphics cards. Plenty of bandwidth available in the new generations of HBM memory makes them more suitable as the last level cache (L4). Due to their layout on the board – HBMs are placed on the same package substrate with CPUs/GPUs, stacked on top of the interposer – providing security features for HBM memory systems imposes lower overheads. Using this

new and large cache memory allows engineers to implement security features more efficiently.

HBMv3[1] may have a logic layer underneath the DRAM stack. This logic layer allows security designers to move security primitives closer to the memory, reducing the memory bandwidth overhead. Exploiting this new smart and large LLC to design more efficient, secure processors should be a new line of research in the security field. For example, an SGX-enabled CPU augmented with HBM memories can establish its entire 128MB PRM – or even larger than that – on the HBM memory. Therefore, during execution time, security checks for PRM accesses will be performed at a lower cost.

- **Flexible TEEs**. Almost all commercial TEEs, except for Intel® SGX, are vulnerable against physical attacks. On the other hand, SGX does not allow users to opt out of physical attack protection. Moreover, none of them provide obliviousness for applications. Current off-the-shelf TEEs provide a fixed number of security features, which gives customers two extreme options: full-fledged security features, which imposes a significant overhead on the application, or opt-out of the all security primitives, which may lead to a serious security/privacy threat. Due to the fact that different applications, even different data structures in one application, have different security requirements, these two options do not suffice for customers. A TEE needs to provide applications with a range of security features, such that developers can define different features for different data structures in their applications. If a commercial TEE such as SGX is intended to provide different security levels, what extra instructions are required? What advantages and disadvantages will this flexibility create? And what is its overhead on the hardware of TEEs?

- **Low Overhead, Industry-Friendly Integrity Tree**. In this thesis, we introduced two new structures for integrity trees to enhance bandwidth overhead. However, there are other techniques that can further improve the bandwidth overhead of integrity verification. For example, during execution time, the working set size of applications may change, and at each time window, they may access part of this working set more frequently. It is

[1]The industry standards of this generation of HBM are not yet officially available

quite reasonable to dynamically update the integrity tree during execution time, based on the current working set of applications. Therefore, as the working set size shrinks, the integrity tree size will decrease, its cachability will increase, and the bandwidth overhead of integrity verification will reduce proportionally.

In the state-of-the-art integrity trees, including the proposals in this thesis, counters are split into global and private counters, where a private counter may overflow, causing other counters in the same block to get reset, which increases the bandwidth overhead. Overflow overhead evaluations in Chapters 4 (Subsection 4.5.2) and 5 (Subsection 5.5.4) show that this overhead for the benchmarks examined in this dissertation, is trivial. However, industry prefers not to add more uncertainty to the design. Hence, they still use a large counter – e.g., SGX uses a 56-bit counter for each data block – to ensure that overflow never occurs. A solution where the width of counters grows dynamically during execution to avoid overflow does not seem practical. Designing an integrity tree resilient against counter overflow, while its overhead is still in check, is a real industrial need to improve commercial TEEs.

- **Low Overhead Obliviousness**. We discussed the research opportunities in the security field created by a large last level cache, provided by HBM memories, previously. However, for obliviousness, enlarging the client storage – in the CPU-memory model, the client storage means the CPU cache, such as the HBM memory – can play a special role in alleviating the bandwidth overhead of Oblivious RAM (ORAM). It is well-known that Path-ORAM [175] has the best bandwidth overhead when the client (CPU) storage is small. The question is if the size of storage in the CPU side grows, is Path ORAM still the winner among different oblivious RAMs? Square Root ORAM [76] is a hierarchical ORAM with $O(1)$ online bandwidth overhead, while it has a significant offline bandwidth overhead for shuffling. In hierarchical ORAMs, the size of client storage, the memory in the CPU side, can reduce the overall overhead of shuffling; besides, Stefano et al. [173], [174] proposed some optimizations to improve the shuffling overhead, reducing the bandwidth cost significantly. Considering the growing size of HBM relative to the external memory, in addition to proposed optimizations, which technique has the lower bandwidth overhead, Path ORAM, or hierarchical ORAM?

Also, what is the lower bound for shuffling overhead? What if we exploit in-memory computation on the DIMM to implement shuffling in the memory side? Combining these ideas, what would be the lowest bandwidth overhead we can achieve to provide obliviousness?

- **Programmable TEEs**. There are multiple hash algorithms: e.g., MD5, SHA-0, SHA-1, SHA-2, or SHA-3; there are multiple algorithms to compute MAC: e.g., CMAC, PMAC, HMAC, UMAC, or VMAC. We can choose a different length for MAC tags, or hash values: e.g., 64b MAC vs. 128b MAC. There are multiple encryption/decryption algorithms: e.g., RSA, DES, or AES, combined with counter-mode encryption. As mentioned above, based on the size of the LLC, we can choose different ORAM algorithms to achieve a lower cost: e.g., Path ORAM vs. hierarchical ORAM. In an integrity tree, the counters' width can be chosen according to the application, and the number of potential overflows the counters may encounter. In memory safety checks, the number of buffers that contain the bound for different pointers can vary according to the application.

  Programmability is a solution that enables the security engine to support this vast range of choices. Therefore, we can have a programmable engine for security purposes that can be customized based on applications' requirements. To that end, we can design a special-purpose processor with a limited, domain-specific ISA, which can be programmed and customized for different security purposes.

- **Designing a More Efficient Metadata Cache.** Our results have shown that caching is an effective technique to reduce the bandwidth overhead of security features. A study can be focused on designing a more effective cache for security purposes. Can a mixture of cache and scratch-pad configuration outperform a pure cache configuration? Cache and scratch-pad configuration is the structure in which a part of the metadata cache works as a scratch-pad, where contains The largest possible level of the integrity tree; the other part still performs as a conventional cache. Therefore, no request for any integrity nodes beyond the level residing in the scratch pad will be issued to the memory system. It is well-known that the replacement policy impacts the cache hit rate. Designing a replacement policy that suits metadata caches the best can improve cache efficiency. Some types of metadata – counters, MAC, and integrity nodes – may show

poor reusability, polluting the metadata cache, which hurts the cache hit rate. Therefore, choosing to cache some types of metadata, while others opt-out may improve metadata cache performance. It is also possible to store victim blocks from the metadata cache in the LLC. It should be studied whether or not the mixture of the LLC and metadata caches can outperform the pure metadata cache configuration.

# REFERENCES


[1] *AMD64 architecture programmer's manual*, AMD Technology. Volume 2, Technical report, 2020, https://www.amd.com/system/files/TechDocs/24593.pdf.

[2] *AMD64 technology indirect branch control extension*, Advanced Micro Device (AMD). White paper, Revision 4.10.18, 2018, https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf.

[3] *ARM security technology building a secure system using trustzone technology*, ARM Limited. 2009. Reference no. PRD29-GENC-009492C.

[4] *Cachegrab*, nccgroup. https://github.com/nccgroup/cachegrab.

[5] *Cloud security solutions forecast, 2018 to 2023 (global)*, Forrester analytics. https://www.forrester.com/report/Forrester+Analytics+Cloud+Security+Solutions+Forecast+2018+To+2023+Global/-/E-RES148715#.

[6] *Intel® digital random number digital random number generator(DRNG)*. Rev. 1.1, 2012.

[7] *Intel® software guard extensions programming reference*, Intel® Corporation. 2014.

[8] *Masked write transfer*. https://www.jedec.org/standards-documents/dictionary/terms/masked-write-transfer-mwt.

[9] *Micron system power calculator*. http://www.micron.com/products/support/power-calc.

[10] *Secure encrypted virtualization API*, Advanced Micro Devices (AMD). Version 0.24, Revision 3.24, 2020.

[11] *Security tip, understanding denial-of-service attacks*, Cybersecurity and Infrastructure security agency. https://us-cert.cisa.gov/ncas/tips/ST04-015.

[12] *SGX security*, Systems Software and Security Lab. https://sgx101.gitbook.io/sgx101/sgx-bootstrap/overview.

[13] *Take control of protecting your data*, Intel® SGX. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[14] *The treacherous 12, cloud computing top threads in 2016*, Cloud Security Alliance. https://downloads.cloudsecurityalliance.org/assets/research/top-threats/Treacherous-12_Cloud-Computing_Top-Threats.pdf.

[15] *U.S. charges Russian hacker with stealing LinkedIn data*, RadioFreeEurope RadioLibrary. https://www.rferl.org/a/us-charges-russian-hacker-nikulin-linkedin-san-francisco-dropbox.

[16] *DDR3 SDRAM part MT41J256M8 datasheet*, 2006. https://datasheetspdf.com/pdf/720199/Micon/MT41J256M8/1.

[17] *Wind River Simics full system simulator*, 2007. http://www.windriver.com/products/simics/.

[18] *Top threats to cloud computing v1.0*, Cloud Security Alliance, 2010. 2010, https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf.

[19] *DDR3 ECC with data mask*, 2013. https://forums.xilinx.com/t5/Other-FPGA-Architectures/ddr3-ecc-with-data-mask/td-p/570028.

[20] *Yahoo! data breaches*, Wikipedia, 2014. https://en.wikipedia.org/wiki/Yahoo!_data_breaches.

[21] *Product change notification*, Intel® Corporation, 2015. https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf.

[22] *Intel® SGX tutorial*, Intel® Corporation, 2015. https://software.intel.com/sites/default/files/332680-002.pdf.

[23] *Cloudbleed: Big web brands "leaked crypto keys, personal secrets" thanks to cloudflare bug*, 2017. https://www.theregister.com/2017/02/24/cloudbleed_buffer_overflow_bug_spaffs_personal_data/.

[24] *Intel® software guard extensions (SGX) developer guide*, Intel® Developer Zone, 2017. https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top.html.

[25] *Security in an ARMv8 system*, ARM, 2017. Reference no. ARM 100935_0100_en, .https://static.docs.arm.com/100935/0100/security_in_an_armv8_system_100935_0100_en.pdf.

[26] *Top threats to cloud computing+industry insights*, Cloud Security Alliance, 2017. https://downloads.cloudsecurityalliance.org/assets/research/top-threats/treacherous-12-top-threats.pdf.

[27] *Extending secure encrypted virtualization with SEV-ES*, 2018. https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Extending-Secure-Encrypted-Virtualization-with-SEV-ES-Thomas-Lendacky-AMD.pdf.

[28] *Top threats to cloud computing, deep dive*, Cloud Security Alliance, 2018. https://cloudsecurityalliance.org/artifacts/top-threats-to-cloud-computing-deep-dive/.

[29] *HBM interface intel® FPGA IP user guide*, 2019. https://www.intel.com/content/www/us/en/programmable/documentation/mhi1462215825912.html.

[30] *AMD SEV-SNP: Strengthening VM isolation with integrity protection and more*, Advanced Micro Device(AMD), 2020. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[31] *Top threads to cloud computing in 2020*, Cloud Security Alliance, 2020. https://cloudsecurityalliance.org/.

[32] S. Aga and S. Narayanasamy, *Invisipage: Oblivious demand paging for secure enclaves*, in ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019, pp. 372–384.

[33] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, *Innovative technology for cpu based attestation and sealing*. 2013, https://software.intel.com/content/www/us/en/develop/articles/innovative-technology-for-cpu-based-attestation-and-sealing.html.

[34] F. Armknecht, A. R. Sadeghi, S. Schulz, and C. Wachsmann, *A security framework for the analysis and design of software attestation*. ACM SIGSAC Conference on Computer & Communications Security (CCS), 2013, pp. 1-12.

[35] S. Arnautov and C. Fetzer, *Controlfreak: Signature chaining to counter control flow attacks*, in IEEE 34th Symposium on Reliable Distributed Systems (SRDS), 2015, pp. 84–93.

[36] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, *SCONE: Secure linux containers with intel® SGX.*, in OSDI, 2016, pp. 689–703.

[37] J. P. Aumasson and L. Merino, *SGX secure enclaves in practice security and crypto review*. https://www.blackhat.com/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-Crypto-Review.pdf.

[38] A. Awad, Y. Wang, D. Shands, and Y. Solihin, *ObfusMem: A low-overhead access obfuscation for trusted memories*, in International Symposium on Computer Architecture (ISCA), 2017, pp. 107-119.

[39] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, *Medrec: Using blockchain for medical data access and permission management*, in 2nd International Conference on Open and Big Data (OBD), 2016, pp. 25–30.

[40] C. Babcock, *93 million mexican voter database exposed on amazon cloud*. 2016, https://www.informationweek.com/cloud/infrastructure-as-a-service/93-million-mexicanvoter-database-exposed-on-amazon-cloud/d/d-id/1325259.

[41] 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, *The NAS parallel benchmarks*, The International Journal of Supercomputer Applications, (1994), pp. 63–73.

[42] R. Barry and D. Volz, *Ghosts in the clouds: Inside China's major corporate hack*. https://www.wsj.com/articles/ghosts-in-the-clouds-inside-chinas-major-corporate-hack-11577729061.

[43] A. Baumann, M. Peinado, and G. Hunt, *Shielding applications from an untrusted cloud with haven*. 11th USENIX Symposium on Operating Systems Design and Implementation, 2014.

[44] A. Baumann, M. Peinado, and G. Hunt, *Shielding applications from an untrusted cloud with haven*, in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 267–283.

[45] S. Beamer, K. Asanović, and D. Patterson, *The gap benchmark suite*, 2017. https://arxiv.org/abs/1508.03619.

[46] M. Benchoufi and P. R. Raphael Porcher, *Blockchain protocols in clinical trials: Transparency and traceability of consent*, 2017. https://pubmed.ncbi.nlm.nih.gov/29167732/.

[47] D. Boneh, R. A. Demillo, and R. J. Lipton, *On the importance of checking computations (extended abstract)*, 1996.

[48] F. Brasser, U. Muller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, *Software grand exposure: SGX cache attacks are practical*, 2017. https://arxiv.org/abs/1702.07521.

[49] J. Bucek, K. Lange, and J. V. Kistowski, *SPEC CPU2017: Next-generation compute benchmark*, in ACM/SPEC International Conference on Performance Engineering, 2018, pp. 41–42.

[50] J. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, *Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution*, in 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, 2017, USENIX Association, pp. 1041–1056.

[51] R. W. Carr and J. L. Hennessy, *WSCLOCK: A simple and effective algorithm for virtual memory management*, in Proceedings of the Eighth ACM Symposium on Operating Systems Principles(SOSP), 1981, pp. 87–95.

[52] D. Champagne, R. Elbaz, and R. B. Lee, *The reduced address space (RAS) for application memory authentication*, in Proceedings of the 11th International Conference on Information Security (ISC), 2008, pp. 47–63.

[53] D. Champagne and R. Lee, *Scalable architectural support for trusted software*, in Proceedings of HPCA, 2010, pp. 1–12.

[54] 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*, 2012. University of Utah.

[55] S. Checkoway and H. Shacham, *Iago attacks: Why the system call API is a bad untrusted rpc interface*, in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 253–264.

[56] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, *SGXPectre: Stealing intel® secrets from SGX enclaves via speculative execution*, in IEEE European Symposium on Security and Privacy (EuroS&P), 2019, pp. 142–157.

[57] H.-M. Chen, C.-J. Wu, T. Mudge, and C. Chakrabarti, *RATT-ECC: Rate adaptive two-tiered error correction codes for reliable 3d die-stacked memory*, in ACM Transaction in Architecture Code Optimization, 2016, pp. 1–24.

[58] S. Chen, X. Zhang, M. Reiter, and Y. Zhang, *Detecting privileged side-channel attacks in shielded execution with déjá vu*, in Proceedings of ACM on Asia Conference on Computer and Communications Security, 2017, pp. 7–18.

[59] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, *SecureME: A Hardware-Software Approach to Full System Security*, in Proceedings of the International Conference on Supercomputing, ACM, 2011, pp. 108–119.

[60] D. Clarke, S. Devadas, M. Van Dijk, B. Gassend, and G. E. Suh, *Incremental multiset hash functions and their application to memory integrity checking*, in International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2003, pp. 188–207.

[61] N. Confessore, *Cambridge Analytica and Facebook: The scandal and the fallout so far*, 2020. The New York Times. ISSN 0362-4331, https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html.

[62] V. Costan and S. Devadas, *Intel® SGX explained*, 2016. https://eprint.iacr.org/2016/086.pdf.

[63] V. Costan, I. Lebedev, and S. Devadas, *Sanctum: Minimal hardware extensions for strong software isolation*, in 25th USENIX Security Symposium (USENIX Security 16), 2016, pp. 857–874.

[64] H. Dai, H. P. Young, T. J. Durant, G. Gong, M. Kang, H. M. Krumholz, W. L. Schulz, and L. Jiang, *Trialchain: A blockchain-based platform to validate data integrity in large, biomedical research studies*, 2018. arXiv, eprint 1807.03662.

[65] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, *Authentic third-party data publication*, in Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security: Data and Application Security, Development and Directions, 2001, pp. 101–112.

[66] T. Dierks and E. Rescorla., *The transport layer security (TLS) protocol version 1.2. RFC 5246 (proposed standard)*, 2008.

[67] M. Dworkin, *Recommendation for block cipher modes of operation: The CMAC mode for authentication*. Federal Information Processing Standards (FIPS) Special Publications (SP), NIST Special Publication 800-38A, 2005.

[68] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, *TEC-Tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks*, in Cryptographic Hardware and Embedded Systems (CHES), 2007, pp. 289–302.

[69] 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, pp. 693–707.

[70] C. Farivar, *Zynga sues 2 former employees over alleged massive data heist*. 2016, https://arstechnica.com/tech-policy/2016/11/zynga-sues-2-former-employees-over-alleged-massive-data-heist/.

[71] 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, pp. 1–12.

[72] C. W. Fletcher, M. v. Dijk, and S. Devadas, *A secure processor architecture for encrypted computation on untrusted programs*, in Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing, 2012, pp. 3–8.

[73] J. Fruhlinger, *Equifax data breach faq: What happened, who was affected, what was the impact?* https://www.csoonline.com/article/3444488/equifax-data-breach-faq-what-happened-who-was-affected-what-was-the-impact.html.

[74] M. Garriga, S. D. Palma, M. Arias, A. D. Renzis, R. Pareschi, and D. A. Tamburri, *Blockchain and cryptocurrencies: A classification and comparison of architecture drivers*, 2020. arXiv, eprint 12283, https://doi.org/10.1002/cpe.5992.

[75] B. Gassend, G. E. Suh, D. E. Clarke, M. van Dijk, and S. Devadas, *Caches and hash trees for efficient memory integrity verification*, in Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA), 2003, pp. 295–306.

[76] O. Goldreich and R. Ostrovsky, *Software protection and simulation on oblivious RAMs*, in J. ACM, Vol. 43, 1996, pp. 431–473.

[77] S.-L. Gong, J. Kim, S. Lym, M. Sullivan, H. David, and M. Erez, *DUO: Exposing On-Chip Redundancy to Rank-Level ECC for High Reliability*, in Proceedings of HPCA, 2018, pp. 683–695.

[78] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, *Cache attacks on intel® SGX*, in Proceedings of the 10th European Workshop on Systems Security, Association for Computing Machinery, 2017.

[79] J. Graham-Cumming, *Incident report on memory leak caused by cloudflare parser bug*, 2017. https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/.

[80] D. Gruss, C. Maurice, and S. M. gard, *Rowhammer. js: A remote software-induced fault attack in javascript*. CoRR, abs/1507.06955, 2015.

[81] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, *Flush+ flush: a fast and stealthy cache attack*, in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2016, pp. 279–299.

[82] S. Gueron, *A memory encryption engine suitable for general purpose processors*. Cryptology ePrint Archive, Report 2016/204, 2016. https://eprint.iacr.org/2016/204.

[83] E. Hall and C. S. Jutla, *Parallelizable authentication trees*, in the 12th International Conference on Selected Areas in Cryptography, 2006, pp. 95—-109.

[84] J. L. Henning, *SPEC CPU2006 benchmark descriptions*, in Proceedings of ACM SIGARCH Computer Architecture News, 2005, pp. 1–17.

[85] R. Huang and G. Suh, *IVEC: Off-chip memory integrity protection for both security and reliability*, in Proceedings of ISCA, 2010, pp. 395–406.

[86] 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, pp. 639–650.

[87] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, *Ryoan: A distributed sandbox for untrusted computation on secret data*, in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI, 2016, pp. 533–549.

[88] J. Jeddeloh and B. Keeth, *Hybrid Memory Cube (HMC) – new dram architecture increases density and performance*, in Symposium on VLSI Technology, 2012, pp. 87–88.

[89] J.-H. Jeng and T.-K. Truong, *On decoding of both errors and erasures of a reed-solomon code using an inverse-free berlekamp-massey algorithm*, in IEEE Transactions on Communications, vol. 47, 1999, pp. 1488–1494.

[90] H. Jeon, G. H. Loh, and M. Annavaram, *Efficient RAS support for die-stacked DRAM*, in 2014 International Test Conference, 2014, pp. 1–10.

[91] 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 the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2013, pp. 1–12.

[92] X. Jian and R. Kumar, *ECC Parity: A technique for efficient memory error resilience for multi-channel memory systems*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2014, pp. 1035–1046.

[93] J.Rodrigues, I.Torre, G.Fernandez, and M.Lopez-Coronado, *Analysis of the security and privacy requirements of cloud-based electronic health records systems*, in Journal of Medical Internet Research, vol. 15, 2013.

[94] J.Szefer and S. Biedermann, *Towards fast hardware memory integrity checking with skewed merkle trees*, in Proceedings of HASP, 2014, pp. 1–8.

[95] D. Kaplan, *Protecting VM register state with SEV-ES*. White paper, 2017, https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf.

[96] J. Kelsey and V. Rijmen, *Compression and information leakage of plaintext*, in Fast Software Encryption, 2002, pp. 263–276.

[97] J. Kim, M. Sullivan, and M. Erez, *Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory*, in the Proceedings of HPCA, 2015, pp. 1–12.

[98] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, *Flipping bits in memory without accessing them: An experimental study of dram disturbance errors*, in Proceeding of the 41st annual International Symposium on Computer Architecture (ISCA), 2014, pp. 361–372.

[99] 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.

[100] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, *SGXBOUNDS: Memory safety for shielded execution*, in Proceedings of the Twelfth European Conference on Computer Systems, EuroSys, 2017, pp. 205–221.

[101] 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, pp. 695–711.

[102] B. Lapid and A. Wool, *Cache-attacks on the arm trustzone implementations of AES-256 and AES-256-GCM via GPU-based analysis*. Cryptology ePrint Archive, 2018. https://eprint.iacr.org/2018/621.

[103] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, *An off-chip attack on hardware enclaves via the memory bus*, in IEEE Symposium on Security and Privacy (SP), 2020, pp. 1–18.

[104] R. B. Lee, *Security basics for computer architects, synthesis lectures on computer architecture*. Morgan Claypool, 2013.

[105] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Zhenghong Wang, *Architecture for protecting critical secrets in microprocessors*, in 32nd International Symposium on Computer Architecture (ISCA), 2005, pp. 2–13.

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

[107] T. S. Lehman, A. D. Hilton, and B. C. Lee, *Poisonivy: Safe speculation for secure memory*, in Proceedings of MICRO, 2016, pp. 1–13.

[108] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, *Architectural support for copy and tamper resistant software*, in Proceedings of International Conference on Architecture Support for Programming Languages and OS(ASPLOS-IX), 2000, pp. 168–177.

[109] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, *Armageddon: Cache attacks on mobile devices*. In USENIX Security conference, pages 549–564, 2016. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_lipp.pdf.

[110] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, *Meltdown*, 2018. https://meltdownattack.com/meltdown.pdf.

[111] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, *Last-level cache side-channel attacks are practical*, in 2015 IEEE Symposium on Security and Privacy, 2015, pp. 605–622.

[112] Logsign Blog, *The biggest cyber-attacks in 2019*. https://blog.logsign.com/the-biggest-cyber-attacks-in-2019/.

[113] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2005, pp. 190–200.

[114] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowic, and D. Song, *PHANTOM: Practical oblivious computation in a secure processor*, in Proceedings of CCS, 2013, pp. 311–324.

[115] MACJournals, *The HFS primer*, 2003. http://macjournal.com/~mwj/mwj_samples/MWJ_20030525.pdf.

[116] A. Malek, E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, *Odd-ECC: On-demand dram error correcting codes*, in Proceedings of MEMSYS, 2017, pp. 96–111.

[117] G. Mappouras, A. Vahid, R. Calderbank, D. R. Hower, and D. J. Sorin, *Jenga: Efficient fault tolerance for stacked DRAM*, in IEEE International Conference on Computer Design (ICCD), 2017, pp. 361–368.

[118] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, *A general model for authenticated data structures*. Springer, 2004, https://doi.org/10.1007/s00453-003-1076-8.

[119] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, *Intel® software guard extensions (intel® SGX) support for dynamic memory management inside an enclave*. Proceedings of the Hardware and Architectural Support for Security and Privacy, 2016.

[120] 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.

[121] MDS: Microarchitectural Data Sampling, *Attacks on the newly-disclosed "MDS" hardware vulnerabilities in Intel® CPUs*. https://mdsattacks.com/.

[122] J. Melnick, *Cloud security risks and concerns in 2018*. https://blog.netwrix.com/2018/01/23/cloud-security-risks-and-concerns-in-2018/.

[123] R. C. Merkle, *Protocols for public key cryptosystems*, in IEEE Symposium on Security and Privacy, 1980, pp. 122–134.

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

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

[126] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, *Fallout: Reading kernel writes from user space*, in arXiv preprint arXiv:1905.12701, 2019.

[127] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, *Oblix: An efficient oblivious search index*, in 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 279–296.

[128] A. Moghimi, G. Irazoqui, and T. Eisenbarth, *Cachezoom: How SGX amplifies the power of cache attacks*, in International Conference on Cryptographic Hardware and Embedded Systems, Springer, 2017, pp. 69–90.

[129] T. Moscibroda and O. Mutlu, *A case for bufferless routing in on-chip networks*, in Proceedings of ISCA, 2009, pp. 196–207.

[130] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, *Plundervolt: Software-based fault injection attacks against intel® SGX*, in Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20), 2020, pp. 1466–1482.

[131] E. Mykletun, M. Narasimha, and G. Tsudik, *Authentication and integrity in outsourced databases*, in ACM Transaction on Storage, 2006, pp. 107–138.

[132] P. Nair, V. Sridharan, and M. Qureshi, *XED: Exposing on-die error detection information for strong memory reliability*, in Proceedings of ISCA, 2016, pp. 341–353.

[133] P. J. Nair, D. A. Roberts, and M. K. Qureshi, *Citadel: Efficiently protecting stacked memory from tsv and large granularity failures*, in 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 51–62.

[134] M. Narasimha and G. Tsudik, *Authentication of outsourced databases using signature aggregation and chaining*, in ACM Transaction on Storage, 2006, pp. 107–138.

[135] M. Neve and K. Tiri, *On the complexity of side-channel attacks on AES-256 – methodology and quantitative results on cache attacks*, 2007. Technical report, https://eprint.iacr.org/2007/318.

[136] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, *Trustzone explained: Architectural features and use cases*, in 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), 2016, pp. 445–451.

[137] 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.

[138] U. G. A. Office, *Data protection: Actions taken by Equifax and federal agencies in response to the 2017 breach*. https://www.gao.gov/products/GAO-18-559,2018.

[139] D. Okeeffe, D. Muthukumaran, P. Aublin, F. Kelbert, C. Priebe, J. Lind, H. Zhu, and P. Pietzuch, *Spectre attack against SGX enclave*. https://github.com/lsds/spectre-attack-sgx.

[140] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, *Eleos: Exitless os services for SGX enclaves*, in EuroSys, 2017, pp. 238–253.

[141] D. Palframan, N. Kim, and M. Lipasti, *COP: To compress and protect main memory*, in Proceedings of ISCA, 2015, pp. 682–693.

[142] H. H. Pang and K. L. Tan, *Authenticating query results in edge computing*, in Proceedings of 20th International Conference on Data Engineering, 2004, pp. 560–571.

[143] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, *Base-delta-immediate compression: Practical data compression for on-chip caches*, in Proceedings of PACT, 2012, pp. 377–388.

[144] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, *DRAMA: Exploiting dram addressing for cross-cpu attacks*, in Proceedings of USENIX Security Symposium, 2016, pp. 565–581.

[145] M. Qureshi, *CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping*, in Proceedings of MICRO, 2018, pp. 775–787.

[146] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas, *Design space exploration and optimization of path oblivious RAM in secure processors*, in Proceedings of ISCA, 2013, pp. 571–582.

[147] Intel® Corporation, *Intel® software developer's manual*. 2015. Reference no. 325462-056US.

[148] J. Robertson and M. Riley, *The big hack: How China used a tiny chip to infiltrate u.s. companies*, 2018. Bloomberg Businessweek, https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies.

[149] 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, pp. 183–196.

[150] M. Sabt, M. Achemlal, and A. Bouabdallah, *Trusted execution environment: What it is, and what it is not*, in IEEE Trustcom/BigDataSE/ISPA, 2015, pp. 57–64.

[151] 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, pp. 416–427.

[152] 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, pp. 454–465.

[153] J. Salowey, A. Choudhury, and D. McGrew, *AES Galois Counter Mode (GCM) cipher suites for TLS*, 2008. https://tools.ietf.org/html/rfc5288.

[154] S. Sasy, S. Gorbunov, and C. W. Fletcher, *Zerotrace : Oblivious memory primitives from Intel® SGX*, in 25th Annual Network and Distributed System Security Symposium (NDSS), 2018, pp. 1–15.

[155] A. Saxena and B. Soh, *Authenticating mobile agent platforms using signature chaining without trusted third parties*, in 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, 2005, pp. 282–285.

[156] A. Saxena and B. Soh, *One-way signature chaining: A new paradigm for group cryptosystems*, in International Journal of Information and Computer Security, 2008, pp. 268–296.

[157] E. Sayegh, *More cloud, more hacks: 2020 cyber threats*. https://www.forbes.com/sites/emilsayegh/2020/02/12/more-cloud-more-hacks-pt-2,2020.

[158] B. Schroeder, E. Pinheiro, and W. D. Weber, *DRAM errors in the wild: A large-scale field study*, in Proceedings of SIGMETRICS, 2009, pp. 100–107.

[159] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, *Malware guard extension: Using SGX to conceal cache attacks*, in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2017, pp. 3–24.

[160] M. Seaborn and T. Dullien, *Exploiting the DRAM rowhammer bug to gain kernel privileges*, 2015. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[161] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li, *Secure DIMM: Moving oram primitives closer to memory*, in Proceedings of HPCA, 2018, pp. 428–440.

[162] 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, pp. 89–101.

[163] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, *MemZip: Exploiting unconventional benefits from memory compression*, in Proceedings of HPCA, 2014, pp. 638–649.

[164] A. Shaizeen and N. Satish, *Invisimem: Smart memory for trusted computing*, in International Symposium on Computer Architecture (ISCA), 2017, pp. 94–106.

[165] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, and Y. Xia, *Occlum: Secure and efficient multitasking inside a single enclave of intel® SGX*, 2001. https://arxiv.org/abs/2001.07450.

[166] Y. Shi, K. Zhang, and Q. Li, *A new data integrity verification mechanism for SaaS*, in Web Information Systems and Mining, Springer Berlin Heidelberg, 2010, pp. 236–243.

[167] M. Shih, S. Lee, T. Kim, and M. Peinado, *T-SGX: Eradicating controlled-channel attacks against enclave programs*, in NDSS Symposium, 2017, pp. 640–656.

[168] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, *Preventing page faults from telling your secrets*, in Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ACM, 2016, pp. 317–328.

[169] R. Sinha, S. Rajamani, and S. Seshia, *A compiler and verifier for page access oblivious computation*, in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, 2017, pp. 649–660.

[170] R. Spreitzer and T. Plos, *Cache-access pattern attack on disaligned aes t-tables*, in International Workshop on Constructive Side-Channel Analysis and Secure Design, Springer, 2013., pp. 200–214.

[171] V. Sridharan and D. Liberty, *A study of dram failures in the field*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2013, pp. 1–11.

[172] 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, pp. 297–310.

[173] E. Stefanov and E. Shi, *Oblivistore: High performance oblivious cloud storage*, in Proceedings of IEEE S&P, 2013, pp. 253–267.

[174] E. Stefanov, E. Shi, and D. Song, *Towards practical oblivious RAM*. arXiv preprint arXiv:1106.3652, 2011, https://arxiv.org/abs/1106.3652.

[175] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, *Path ORAM: An extremely simple oblivious ram protocol*, in Proceedings of CCS, 2013, pp. 1–26.

[176] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, *Efficient memory integrity verification and encryption for secure processors*, in Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, pp. 1–12.

[177] G. E. Suh and S. Devadas, *Design and implementation of the AEGIS single-chip secure processor using physical random functions*, in Proceedings of ISCA, 2005, pp. 570–580.

[178] P. Szalachowski, D. Reijsbergen, I. Homoliak, and S. Sun, *Strongchain: Transparent and collaborative proof-of-work consensus*, in 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, Aug. 2019, USENIX Association, pp. 819–836.

[179] J. Szefer and M. Martonosi, *Principles of secure processor architecture design*. Morgan Claypool, 2018.

[180] M. Taassori, A. Nag, K. Hodgson, A. Shafiee, and R. Balasubramonian, *Memory: The dominant bottleneck in genomic workloads*, in Proceedings of AACBB Workshop, in conjunction with HPCA-24, 2018.

[181] M. Taassori, A. Shafiee, and R. Balasubramonian, *VAULT: Reducing paging overheads in SGX with efficient integrity verification structures*, in Proceedings of ASPLOS, 2018, pp. 665–678.

[182] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, *Switchless calls made practical in intel® SGX*, in Proceedings of the 3rd Workshop on System Software for Trusted Execution, 2018, pp. 22–27.

[183] H. Tian, Y. Zhang, C. Xing, and S. Yan, *Sgxkernel: A library operating system optimized for intel® SGX*, in Proceedings of the Computing Frontiers Conference, 2017, pp. 35–44.

[184] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, *Cooperation and security isolation of library oses for multi-process applications*, in Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 1–14.

[185] C.-C. Tsai, D. E. Porter, and M. Vij, *Graphene-SGX: A practical library os for unmodified applications on SGX*, in 2017 USENIX Annual Technical Conference (USENIX ATC), 2017, pp. 645–658.

[186] 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, pp. 285–296.

[187] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, *Foreshadow: Extracting the keys to the intel® SGX kingdom with transient out-of-order execution*, in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 991–1008.

[188] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, *Ridl: Rogue in-flight data load*, in 2019 IEEE Symposium on Security and Privacy (SP), IEEE, 2019, pp. 88–105.

[189] 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, pp. 1–7.

[190] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. Gunter, *Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX*, in Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017, pp. 2421–2434.

[191] Y. Wang, A. Ferraiuolo, and G. E. Suh, *Timing channel protection for a shared memory controller*, in HPCA, 2014, pp. 225–236.

[192] Y. Wang, B. Wu, and G. Suh, *Secure dynamic memory scheduling against timing channel attacks*, in Proceedings of HPCA, 2017, pp. 301–312.

[193] Z. Wang and R. B. Lee, *New cache designs for thwarting software cache-based side channel attacks*, in Proceedings of ISCA, 2007, pp. 494–505.

[194] ——, *A novel cache architecture with enhanced performance and security*, in Proceedings of MICRO, 2008, pp. 83–93.

[195] Wangyuan Zhang and Tao Li, *Microarchitecture soft error vulnerability characterization and mitigation under 3d integration technology*, in 2008 41st IEEE/ACM International Symposium on Microarchitecture, 2008, pp. 435–446.

[196] M. N. Wegman and J. Carter, *Universal classes of hash functions*, in Journal of Computer and System Sciences, Vol. 18, 1979, pp. 143–154.

[197] M. N. Wegman and J. lawerence, *New hash functions and their use in authentication and set equality*, in Journal of Computer and System Sciences, 1981, pp. 265–279.

[198] O. Weisse, V. Bertacco, and T. Austin, *Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves*, in Proceedings of the 44th Annual International Symposium on Computer Architecture, ACM, 2017, pp. 81–93.

[199] Y. Xu, W. Cui, and M. Peinado, *Controlled-channel attacks: Deterministic side channels for untrusted operating systems*, in Proceedings of IEEE Symp. on Security and Privacy (S&P Oakland), 2015, pp. 640–656.

[200] 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, pp. 332–345.

[201] Y. Yarom and K. Falkner, *Flush+Reload: A high resolution, low noise, l3 cache side-channel attack*, in the 23rd USENIX Security Symposium, 2014, pp. 719–732.

[202] 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, pp. 403–415.

[203] D. Yoon and M. Erez, *Memory mapped ECC: Low-cost error protection for last level caches*, in Proceedings of ISCA, 2009, pp. 116–127.

[204] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, *Truspy: Cache side-channel information leakage from the secure world on arm devices*. IACR Cryptology ePrint Archive, 2016.

[205] Y. Zhou, S. Wagh, P. Mittal, and D. Wentzlaff, *Camouflage: Memory traffic shaping to mitigate timing attacks*, in High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on, IEEE, 2017, pp. 337–348.