A Memory Encryption Engine Suitable for General Purpose Processors

Shay Gueron[1,2]

[1]Intel Corporation, Intel Development Center, Israel
[2]University of Haifa, Israel

Version: February 25, 2016-08:10:42

Abstract

Cryptographic protection of memory is an essential ingredient for any technology that allows a closed computing system to run software in a trustworthy manner and handle secrets, while its external memory is susceptible to eavesdropping and tampering. An example for such a technology is Intel's emerging Software Guard Extensions technology (Intel SGX) that appears in the latest processor generation, Architecture Codename Skylake. This technology operates under the assumption that the security perimeter includes only the internals of the CPU package, and in particular, leaves the DRAM untrusted. It is supported by an autonomous hardware unit called the Memory Encryption Engine (MEE), whose role is to protect the confidentiality, integrity, and freshness of the CPU-DRAM traffic over some memory range. To succeed in adding this unit to the micro architecture of a general purpose processor product, it must be designed under very strict engineering constraints. This requires a careful combination of cryptographic primitives operating over a customized integrity tree that mostly resides on the DRAM while relying only on a small internally stored root. The purpose of this paper is to explain how this hardware component of SGX works, and the rationale behind some of its design choices. To this end, we formalize the MEE threat model and security objectives, describe the MEE design, cryptographic properties, security margins, and report some concrete performance results.

1 Introduction

Attacks on the system memory (DRAM) of a computing device are a serious threat. It has been shown that an attacker with physical access to a platform and the right tools, can read and/or modify memory contents. Such capabilities can be used for reading secrets that reside in memory, or for circumventing the platform's security or policies. These threats led to an intensive study of the memory protection problem, for which various elegant solutions have been proposed in the literature (e.g., [6, 8, 13, 14, 22, 17, 20, 21, 24, 25, 26, 10]).

In general, the goals of memory protection include data confidentiality, data integrity, and data replay prevention, but in some scenarios a subset of these goals suffices. For example, a "cold boot attack" is a passive attack that harvests a (single) snapshot of the DRAM contents from a locked device. It can be mitigated by encrypting the memory (for confidentiality). On the other hand, active attacks that use memory modifications, may require a higher bar protection of at least adding integrity checks. Intel's new Software Guard Extensions [1] (SGX hereafter) [3] is a technology that requires full DRAM protection for confidentiality, integrity, and anti-replay.

SGX is designed to allow a general purpose computer platform to run software in a trustworthy manner and to handle secrets. The assumed adversary has full control of the system, and the software running on it at any privilege level, and can read or modify the contents of the DRAM

[1] Formally: Intel® Software Guard Extensions,

(including copy-and-replay). The only components that SGX trusts are the CPU internals. To illustrate why SGX needs full protection for (part of) the DRAM, we provide a brief outline of this technology (see [3], [4], [16], [12] for details).

SGX consists of a set of CPU instructions, supported by a hardware-based access control mechanism. They provide a way to load application code and data from memory, while incrementally locking it in a dedicated DRAM region, and generating its cryptographic measurement (SHA-256 digest). After the code is loaded, it can run in a special mode as a secure "enclave", remaining isolated from all other processes on the system (isolation is governed by the access control mechanism). Note that any application code is merely a binary file (written in some special format). As such, it cannot be shipped with secrets that are already embedded in the file's text. To this end, SGX offers a mechanism for a secret owner to *provision* a secret to a trustworthy enclave. Its instructions provide tools with which, together with an attestation protocol, an enclave can prove to an off-platform entity (or to another enclave on the same platform) that:

a. It is running on a genuine processor under the SGX restrictions.

b. The value it reports for its cryptographically measured identity is trustworthy.

A secret owner can therefore establish a trusted channel with an identified enclave, and provision a secret. For handling provisioned secrets, SGX offers instructions that allow an enclave to encrypt any information with a secret key that is unique to the platform and to its identity. Thus, an enclave can safely store (and retrieve) secrets on untrusted locations (e.g., a disk). Clearly, SGX requires memory protection. Handling secrets requires that the confidentiality of CPU-DRAM traffic (within the relevant regions) is protected. In addition, in order to trust an enclave's cryptographic identity, its intended execution flow, and the attestation protocol, the DRAM (at least part of) must be tamper resistant, and replay protected. We point out that replay prevention is critical. Otherwise, an attacker can select a chunk of code (plus the integrity tags) of a properly loaded enclave, and replace it with a recorded image from a previously loaded forged enclave. The forged enclave can therefore run while the CPU reports the identity of the legitimate enclave.

This paper describes the hardware component of SGX, called the Memory Encryption Engine (MEE), which delivers the required protection for the DRAM. Like similar memory encryption technologies, its design is based on the following pillars: an integrity tree, the cryptographic primitives that realize the encryption, the Message Authentication Code (MAC), and the anti-replay mechanism.

An integrity tree is necessary because internal (on-die) storage is limited and expensive. This requires that the system seizes a dedicated DRAM region where the tree can be stored and maintained. Different integrity trees use various combinations of counters and MAC tags (or hashes), offering tradeoffs between the size of the internal storage and the "seized" DRAM region, the cost/complexity of the "tree verify" and "tree update" flows, and the resulting performance (good reviews comparing several integrity trees are given in [7] and [11]). The choice of the cryptographic primitives is critical for achieving the adequate level of security while satisfying the engineering constraints. These call for a parallelizable encryption mode, and a MAC algorithm that produces short MAC tags with a cheap hardware implementation. The ability to share hardware elements between the encryption and the MAC generation is, of course, also desirable.

Addressing the lean resources of real systems, and the high complexity of integrating a fully autonomous hardware unit in the processor, mandates aggressive optimization of these design pillars. In this paper, we detail the MEE design that turned out to be sufficiently practical to be part of Intel's latest processor generation, Architecture Codename Skylake [2]. The paper is organized as follows. Section 2 details the security model, objectives, and the adversary model. Section 3 describes the rationale and the design of the MEE integrity tree. Section 4 discusses the cryptographic functions and their properties. Some notes on the implementation and optimizations techniques, are brought in Section 5. We challenge the security margins of the MEE design in Section 6, and report some perfor-

[2] The 6th Generation Intel® Core™ processor.

mance results in Section 7.

2 The MEE security model and objectives

MEE system overview. A modern processor has an internal cache that accommodates a small amount of memory, and can be accessed much faster than the system memory. During normal operation, memory transactions are continuously issued by the processor's Core, and transactions that miss the cache are handled by the Memory Controller (MC). The MEE operates as an extension of the MC, taking over the cache-DRAM traffic that points to what is called the "Protected" data region. An additional portion of the memory, called the "seized" region, accommodates the MEE's integrity tree. The union of these regions is called the "MEE region". It forms a range of physical addresses that is fixed to some size at boot time (the default size is 128MB), in a trustworthy way.

Read/write requests to the protected region are routed by the MC to the MEE that encrypts (decrypts) the data before sending (fetching) it to (from) the DRAM. The MEE initiates autonomously additional transactions to verify (update) the integrity tree, based on a construction of counters and MAC tags. The self-initiated transactions access the seized region on the DRAM, and also some on-die SRAM array that serves as the root of the tree. Figure 1 shows a schematic illustration of the MEE operations.

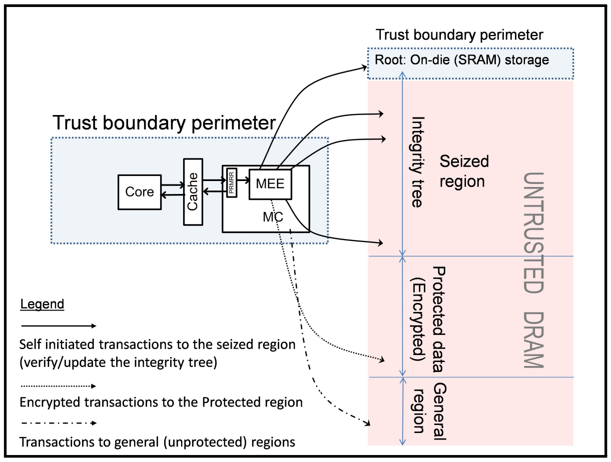
Trust boundary perimeter
Root: On-die (SRAM) storage
Trust boundary perimeter
Core
Cache
PRMRR
MEE
MC
Integrity tree
Seized region
UNTRUSTED DRAM
Protected data (Encrypted)
General region
Legend
Self initiated transactions to the seized region (verify/update the integrity tree)
Encrypted transactions to the Protected region
Transactions to general (unprotected) regions

Figure 1: A schematic illustration of the MEE operation.

Capturing the DRAM changes over time. In our architecture, the MC (and MEE) read and write transactions are at the granularity of 512-bit blocks, called Cache Lines (CL hereafter). If the DRAM is viewed (logically) as an array of CL's indexed by a per-CL address, we can describe its changes over time by a discrete sequence of snapshots, where only one CL is modified between two consecutive snapshots, as follows. Suppose that the DRAM has N CL's at addresses $x_1,\ldots,x_N$. For time steps $y = 0,1,\ldots$, we use the notation $CL(x,y)$ for the contents of the CL at address x, at time y. A snapshot is a full DRAM contents at time y: $Snapshot(y) = \{CL(x_k,y), k = 1,\ldots,N\}$. The information on the DRAM contents over time can be captured by a sequence of snapshots $Snapshot(y)$, $y = 0,1,\ldots$, where $Snapshot(y+1)$ differs from $Snapshot(y)$ in only one location (address).

2.1 The MEE adversarial model

The MEE has two security objectives:

Objective 1. *Providing confidentiality for the data that is written to the Protected region (on the DRAM).*

Objective 2. *Data integrity with replay prevention, assuring that data which is read back from the DRAM's Protected region to the CPU, is the same data that was most recently written from the CPU to the DRAM.*

Remark 1. *The MEE is not designed to be an Oblivious RAM. An adversary with the assumed ability to track DRAM changes over time, can, by definition, carry out traffic analysis. He can learn when CL's are written, and to which CL addresses (though the contents of this traffic remains confidential). Preventing such analysis is not an objective of the MEE.*

The MEE encrypts and computes authentication tags with the following properties.

Property 1. *The MEE keys are generated uniformly at random at boot time, and never leave the die.*

Property 2. *The encryption keys and the authentication keys are separate.*

Property 3 (Drop-and-lock policy)**.** *Tree verifications (and updates) enforce the following "drop-and-lock" policy. The MEE computes the MAC tags of data that it reads,* *and compares them to expected values, fetched from the integrity tree on the DRAM. If all comparisons match, the operation continues. However, as soon as any mismatch is detected, the MEE emits a failure signal, drops the transaction (i.e., no unverified data ever reaches the cache) and immediately locks the MC (i.e., no further transactions are serviced). This causes the system to hang, and it needs to be re-booted. After re-boot, the MEE starts over with newly generated keys.*

The MEE adversaries. The MEE faces two types of adversaries: a *passive* eavesdropping adversary, and an *active* forging adversary. Both adversaries start from observing DRAM snapshots over some period of time, and recording a DRAM trace $\mathscr{T} = Snapshot(y)$, $y = 0,\dots,\tau_{end}$ for some time τ_{end} ($\tau_{end} = 0$ means that the adversary chooses to collect no such information). $\mathscr{T}$ is produced by either: a) enclaves that write their (known or secret) data; b) a crafted attack enclave code that writes chosen plaintext/messages to the Protected region; c) self initiated MEE transactions. An astute adversary tries to manipulate the system in order to generate the most favorable trace for his intended attack.

The passive adversary uses $\mathscr{T}$ to extract a list, $\mathscr{T}'$ of q' plaintext-ciphertext (and associated nonces from the integrity tree) samples. Subsequently, he can apply any algorithm to $\mathscr{T}'$, to obtain information on a targeted secret. Here, a secret is ciphertext from an unknown plaintex, that was written to the DRAM, at least once, by a "victim" enclave.

The active adversary uses $\mathscr{T}$ to extract a list, $\mathscr{T}''$ of q'' message-tag (and nonce) samples. He can use $\mathscr{T}''$ with any algorithm. Subsequently, he modifies the current DRAM snapshot (on the Protected region) in any way he chooses. The attack succeeds if the modified data passes the integrity check when it is read back. From Property 3 it follows that the adversary has only one chance to try a (failing) forgery against the MEE with the same set of keys. A particular form of forgery is a replay attack. A replay is carried out by overwriting (part of) a current snapshot $Snapshot(\tau_2)$ with (part of) a previous (recorded) $Snapshot(\tau_1)$, for some $\tau_1 < \tau_2$. In the presence of a valid integrity tree with a trusted root, a replay attack is not different from any other DRAM modification, unless some values (counters) at time τ_2 repeat their values at time τ_1. This scenario needs to be prevented by the MEE design.

We can now translate the adversarial model to a cryptographic scenario: the passive adversary views the MEE as an encryption oracle, and the active adversary views it as a tagging oracle (and a verification oracle when actually attempting the forgery). By Property 2, the oracles are independent. We assess the MEE's strength against its adversaries by combining two criteria.

Criterion 1: Cryptographic bounds on:

1. The adversary's advantage in distinguishing the outputs of the encryption oracle from a random output, using information from q' *chosen* oracle queries.
2. The adversary's probability to succeed in the *first* forgery attempt, using information gathered from q'' *chosen* queries to a tagging oracle.

Criterion 2: Lower bounds on the time required for

1. Collecting q' (q'') samples.
2. The rate that successive "boot and forgery" attempts can be executed (i.e., repeating forgery attempts until (hopefully) the first success).
3. Exhausting the integrity tree by executing enough tree writes and updates that propagate the counters, until one of them repeats a value.

Our goal is to analyze the MEE properties even against an idealized MEE adversary who is limited only by the physical capacity of the hardware. Therefore, we intentionally grant the adversary much more power than practically feasible. For example, Criterion 1 allows *chosen* input queries (although self initiated transactions are controlled by the MEE). Criterion 2 ignores the overhead of extracting the relevant information $\mathscr{T}'$ / $\mathscr{T}''$ from observing $\mathscr{T}$ (not every system operation translates directly to a useful oracle query). It also ignores the time/cost/complexity of storing at least $\mathscr{T}'$ and $\mathscr{T}''$ (if not the full $\mathscr{T}$) to facilitate offline analysis.

3 The MEE integrity tree

An integrity tree is a standard method for using a small amount of internal storage to protect a larger amount of data (see [7] for an overview). The classical method is the Merkle Tree (MT): a binary tree where each node holds a hash digest of its two children, and the lowest leaves hold digests of the protected data units. Its obvious generalization is the *k*-ary MT with *k* children for each node. For a memory encryption technology that can use an internal integrity key, using MAC tags is typically more efficient than using hashes. Suppose that a stateful MAC algorithm is used, where the nonce is the state, and the MAC covers the nonce and the data units. In such cases, it suffices to protect (only) the nonces by the integrity tree. Making the tree tamper resistant is achieved by storing (only) its root on-die.

The MEE integrity tree uses a stateful MAC with nonces (see also [19]), and its construction is tailored to optimally fit the architecture. We describe the concept by a simplified model in Section 3.1, and the details of the real implementation are given in Section 3.2.

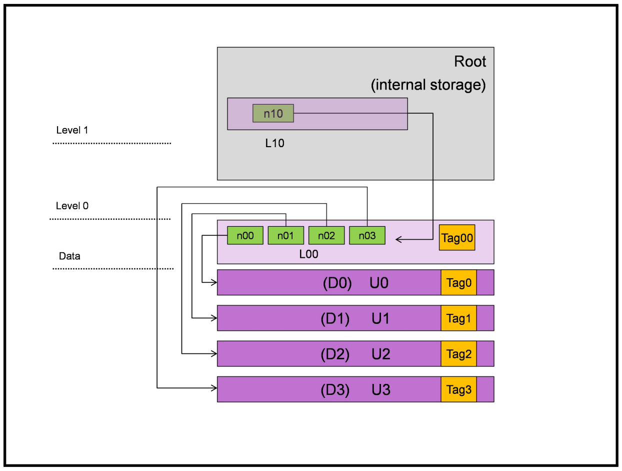

Figure 2: A simplified two levels integrity tree with a 4-ary layout.

3.1 A simplified tree model

Figure 2 shows a simplified integrity tree with two levels (Level 0, Level 1) and a 4-ary layout. The region that is "covered" by the tree consists of data blocks D0, D1, D2, D3 which are stored in data units U0, U1, U2, U3, together with their respective authentication tags Tag0, Tag1, Tag2, Tag3. Tagj (j=0, 1, 2, 3) is computed by some MAC algorithm, over Dj and the nonce n0j. At Level 0, the data unit L00 includes the nonces n00, n01, n02, n03 and the associated authentication tag Tag00 that is computed over (n00, n01, n02, n03) and the nonce n10. The nonce n10 is part of the data unit L10 at Level 1, which is the root of the tree.

In general, the data units are not necessarily of the same size or internal layout, and the MAC algorithm(s) can be applied in different ways. The nonces (not necessarily with the same lengths) are stored as bit strings, viewed as a realization of some counter, where each specific implementation defines how a counter is incremented. Note that the MAC tags are embedded in their respective data unit. We use the following examples to illustrate the "read and verify" and the "write and update" flows.

Read and verify example. Consider reading data block D0, and verifying that it was not modified from its value since the last time it was written and the tree was updated. This requires fetching the data units U0, L00, and L10, and extracting D0, Tag0, n00 and n10 from them. By definition, n10 is trusted. The nonces (n00, n01, n02, n03) are verified (together, as one entity) by applying the MAC algorithm over (n00, n01, n02, n03) and n10, and comparing to Tag00. Specifically, this verifies n00 (i.e., that n00 was not changed since it was written with the proper the tree update). D0 is verified by applying the MAC algorithm over D0 and n00, and comparing it to Tag0. Note that the implementation can choose the order by which U0, L00 and L10 are fetched, the order by which D0, Tag0, n00 and n10 are extracted, and the order by which the expected tags are computed and compared. The only requirement is that D0 is "approved" only if (and after) all the tag comparisons pass.

Remark 2. *Trusting only the internally stored root of the tree, that cannot be modified by an adversary, is sufficient. For example, an adversary can modify D0, Tag0 and n00 by replaying previous values, so that MAC over D0 and n00 passes a comparison to Tag0. However, the forgery would succeed only if the tag over the modified L00 with the unmodified nonce n10 equals Tag00. This means that successful forgery needs to pass the MAC tag check with an unmodified nonce for at least one location on the tree.*

Write and update example. Suppose that D2 needs to be written, with the proper update of the integrity tree. The procedure must start with a preemptive verification of D2 (using n02 and comparing to Tag2). This requires the preemptive verification of n02 (using (n00, n01, n02, n03) and n10, and comparing to Tag00). Then, D2, is modified to D2', n02 is incremented to n02' and n10 to n10'. Tag2 and Tag00 are recomputed by applying the MAC algorithm to D2' and n02', and to (n00, n01, n02', n03) and n10'. Finally, the tree is updated with their new values Tag2' and Tag00'.

Remark 3. . *The preemptive check is crucial. Without it, n03 can be modified to n03', D3 to D3', and Tag3 to Tag3', while D2 is modified to D2'. Then, Tag00' will be computed over (n00, n01, n02', n03'). This allows a successful replay of previous D3, Tag3, n03 values.*

Self initialization. The preemptive checks cause an inevitable MAC mismatch in the first time that a data unit is written. An exception is therefore needed. This is done by: a) assigning a special nonce value (n_init) as an indicator to the fact that a data unit has not been written yet; b) actively setting (only) the internal Level 1 nonces to the value n_init, at system initialization, in a trustworthy way; c) configuring the integrity checker logic to ignore a MAC mismatch when encountering n_init at some uninitialized data unit, and to assign the value n_init to the other nonces of its "children". In our example, n10 is set to n_init at start time. When D2 is written for the fist time, the failing preemptive check is ignored, a "regular" value is assigned to n02, and n_init is assigned to n00, n01, n03.

Non-repeating compound nonces. The tree updating must assure that nonces do not repeat. One way to generate non-repeating nonces is to maintain one *internal* "central" counter. When a fresh nonce is needed for some data unit, the system uses the available counter value, and immediately increments it. The MEE uses an alternative "compound nonce" construction. It includes: a) a unique "spatial coordinate" (address) of the data unit; b) a per-unit counter which is a "temporal coordinate". When a data unit is written, the system uses the available temporal counter value, and then increments it. Here, the temporal counters may repeat, but on different addresses, and this guarantees that the compound nonces remain non-repeating.

Exhausting the tree. A w-bit nonce can be incremented (from n_init) at most $2^w - 1$ times without repeating a value. Thus, the integrity tree can remain "valid" (i.e., providing protection) only up to the first time that a counter will rollover by incrementing one more time. At that point, we say that the tree is exhausted, and the system should drop-and-lock in order to start over with new keys.

Generalizing the construction to a larger number of data units, levels, and different k-ary layouts is straightforward.

3.2 The MEE data structure

Version counters and PD_Tags. Since an adversary should not gain information on unknown plaintexts by observing ciphertext samples on the DRAM, equal plaintext blocks that are encrypted to different addresses and/or at different points in time, should not translate to equal ciphertext blocks. This requires the encryption scheme to use a non repeating location-time tweak. The MEE uses a compound nonce where the spatial coordinate of a unit is its address, and its temporal coordinate is a dedicated per-unit counter. These counters are called "versions" (to distinguish from the counters of the integrity tree).

The integrity tree covers the versions. Each data unit in the "Protected data" region has a MAC tag (PD_Tag) that is computed over the ciphertext and its tweak. The PD_Tags and the versions are stored in a DRAM region called "Metadata". It is enough to have the integrity tree "cover" either the PD_Tag's or the versions. We chose to cover the versions.

The MEE data structure. The specific instantiation of the MEE data structure is defined by setting equal size data units of 512 bits (CL's), counters/versions with w bits, and tags with t bits. Since the Protected data CL's have no architecturally visible redundancy, their PD_Tags and versions are accommodated in the dedicated Metadata region. By contrast, we can leverage some redundancy for the CL's that hold the versions and the counters: a t-bit tag can be embedded in a CL that accommodates n_c w-bit counters, if $t + n_c \cdot w \leq 512$. The parameters choice $n_c = 8$, $w = 56$, $t = 56$ leads to a convenient implementation (Section 6 explains why its security bounds are sufficient). An 8-ary tree can be built over the version CL's. With $n_c \cdot t < 512$, the PD-Tags can also be placed in an 8-ary layout. The parametrized design supports $32, 64, 128, 256$ MB, for the MEE region, 4 or 6 levels, and an 8-ary integrity tree layout. The specific instantiation we report here uses 4 levels, with a root (Level 3) stored on-die (internal SRAM array). Fig. 3 illustrates this construction. The top panel outlines the layout of the MEE region, and the chain of dependencies. The bottom panel zooms into a CL, showing how a 56-bit tag is embedded alongside with 8 counters.

The CL's in the MEE region are stacked in groups of 8, to leverage the 8-ary layout. To read and verify a CL at address $Addrs$, the MEE derives sub-address (pointers) to the CL's that hold the corresponding PD_Tag, version, and counters, and also pointers to the specific field in these CL's. For simplicity, the base address of the region is naturally aligned (to its total size). In addition, "Reserved"gaps are introduced in order to naturally align the sub-regions. This alignment leads to simple bit-level expressions for the derived sub-addresses and sub-fields, which are easily implemented in hardware. Table 1 shows a detailed example of a 128 MB region, and the bit-level expression for the derived pointers (from a 40 bits address). This layout offers 96MB of user-available Protected data, supported with (only) 4KB of internal storage. In general, our construction has $\sim 3/4$ of the MEE region available to the user, with a root (internally stored) of size $\sim 1/2^{15}$ the size of the user-available region.

Counters definition. Counters (and versions) are 56-bit strings that we view as elements of $\mathbb{F}_{2^{56}} = \mathbb{F}(2^{56})/(x^{56} + x^{55} + x^{35} + x^{34} + 1)$. We define n_init to be the the field's multiplicative unit ($0^{55}1$). The root (Level 3 in Figure 3) is initialized to n_init by the hardware, as part of the MEE initialization. Incrementing $A \in \mathbb{F}_{2^{56}}$ is the field operation $INCREMENT(A) = A \times x$ (where x is represented by the string $0^{54}10$). Since x is a primitive element in $\mathbb{F}_{2^{56}}$, n_init can be incremented successively $2^{56} - 2$ times without repeating a value (i.e., exhausting the counter).

Property 4. *Let $\mathscr{T}$ be a DRAM trace where no counter or version is exhausted, and consider the collection of all address-version and address-counter pairs in $\mathscr{T}$. Then, all of these pairs are unique.*

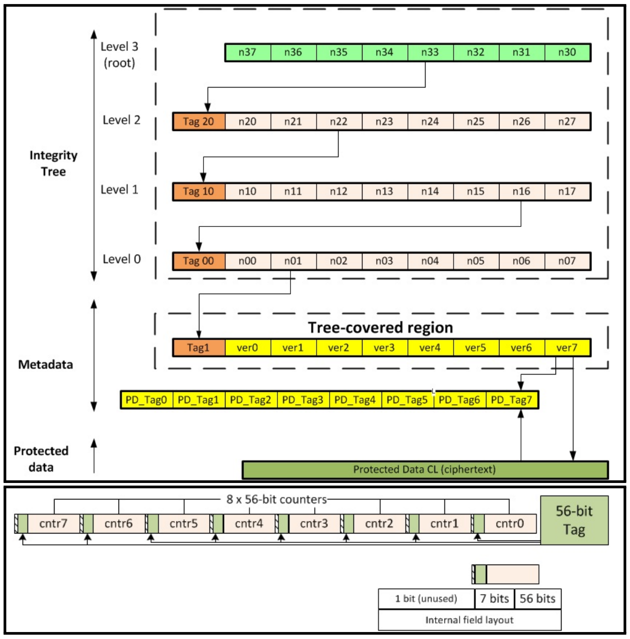

Figure 3: Top panel: the MEE data structure. Bottom panel: embedding a 56-bit tag in a CL that accommodates 8 56-bit counters. The tag is split to 8 7-bit chunks that are distributed (together with one unused bit) in 8 positions adjacent to the counters, across the CL.

4 The MEE cryptographic functions and properties

4.1 Preliminaries

Approximating AES as a random permutation. Let $AES_K(P)$ denote the ciphertext that AES128 generates by encrypting P with the key K. For an integer q, let $\varepsilon_{AES}(q)$ denote the chance of an adversary to distinguish AES ciphertexts from the outputs of a random permutation, after observing q samples from chosen plaintexts (with the

Table 1: Top table: the detailed layout of a 128MB MEE region. Bottom table: The bit-level addressing scheme for the sub-regions and field in a CL.

	Start offset	End offset	Region size
Protected data	000_0000	5ff_ffff	96MB
Version + PD_Tag	600_0000	77f_ffff	24MB
Reserved	780_0000	7df_ffff	6MB
L0	7e0_0000	7f7_ffff	1.5MB
Reserved	7f8_0000	7fb_ffff	256KB
L1	7fc_0000	7fe_ffff	192K
Reserved	7ff_0000	7ff_7fff	32KB
L2	7ff_8000	7ff_dfff	24K
Reserved	7ff_e000	7ff_efff	4KB
L3 (On-die root)	7ff_f000	7ff_ffff	4KB
Total MEE region			128MB

Region	Address Format and Mapping	Field in CL
Data	Addrs[39:6]	
PD_Tag	Addrs[39:27] & ’11 & Addrs[26:9] & ‘0	!Addrs[8:6]
Version	Addrs[39:27] & ’11 & Addrs[26:9] & ‘1	Addrs[8:6]
L0	Addrs[39:27] & ’111111 & Addrs[26:12]	Addrs[11:9]
L1	Addrs[39:27] & ’111111111 & Addrs[26:15]	Addrs[14:12]
L2	Addrs[39:27] & ’111111111111 & Addrs[26:18]	Addrs[17:15]
L3	Addrs[39:27] & ’111111111111111 & Addrs[26:21]	Addrs[20:18]

secret key selected uniformly at random). A standard assumption (AES design goals and analysis [15] [18]) is that $\varepsilon_{AES}(q)$ is very small, even for $q = 2^{64}$ (i.e., beyond the birthday bound). We adopt this assumption in order to approximate AES as a random permutation.

The MEE keys. The MEE initialization generates 768 uniform random bits which are distributed to three (independent) keys:

1. A 128-bit confidentiality key, K_{ENC} (used for data encryption).
2. A 128-bit masking key, K_{MAC} (used for a the MAC algorithm).
3. A 512-bit hash key, viewed as the concatenation $K_7, K_6, \ldots, K_0$, of 8 64-bit strings (used for selecting a function from a universal family of hash functions).

4.2 Carter-Wegman MAC's in the context of the MEE attack scenario

Denote $\mathsf{D}_1 = \{0,1\}^{d_1}$, $\mathsf{R} = \{0,1\}^t$ $(d_1 > t)$, $\mathsf{K}_1 = \{0,1\}^{s_1}$, $\mathsf{D}_2 = \{0,1\}^{d_2}$, $\mathsf{K}_2 = \{0,1\}^{s_2}$. Let $\mathsf{H} = \{h_{k_1} : \mathsf{D}_1 \to \mathsf{R},\ k_1 \in \mathsf{K}_1\}$ be a family of functions indexed by a key $k_1 \in \mathsf{K}_1$. H is called ε Almost XOR Universal (ε-AXU) if $\Pr_{k_1 \leftarrow \mathsf{K}_1}\left(h_{k_1}(\mu) \oplus h_{k_1}(\nu) = \xi\right) \leq \varepsilon$ for every $\xi \in \mathsf{R}$ and $\mu \neq \nu \in \mathsf{D}_1$, where k_1 is selected uniformly at random from the key domain K_1. In case $\varepsilon = 2^{-t}$, H is called XOR-universal. Let $f_{k_2} : \mathsf{D}_2 \to \mathsf{R}$ be a secure PRF selected with a secret key $k_2 \in \mathsf{K}_2$. The maximum q-interpolation probability of f_{k_2} is the maximum probability that $(f_{k_2}(b_1),\ f_{k_2}(b_2),\ \ldots,\ f_{k_2}(b_q)) = (a_1, a_2, \ldots, a_q)$, for all $a_1, a_2, \ldots a_q \in \mathsf{R}$ and distinct $b_1, b_2, \ldots b_q \in \mathsf{D}_2$, where k_2 is selected uniformly at random from K_2.

The Carter-Wegman MAC [23] uses an ε-AXU family H and a PRF f_{k_2} with maximum q-interpolation probability less or equal δ. It is instantiated by selecting uniform random secret keys $k_1 \in \mathsf{K}_1$ and $k_2 \in \mathsf{K}_2$, and operates on a message $M \in \mathsf{D}_1$ and a nonce $b \in \mathsf{D}_2$. The MAC tag is

$$T = h_{k_1}(M) \oplus f_{k_2}(b) \qquad (T \in \mathsf{R}). \tag{1}$$

An adversary submits to a tagging oracle, $q \leq 2^t - 2$ queries, each one with a message M_i with nonce b_i, such that all the nonces are distinct. The oracle replies with the respective tags T_i, $i = 1, \ldots, q$. In the MEE context, the adversary attempts a forgery of the following form:

> Choosing some location $1 \leq j \leq q$, and guessing a message-tag pair pair M_*, T_* such that $(M_*, T_*) \neq (M_j, T_j)$. The forgery attempt is considered successful if $h_{k_1}(M_*) \oplus f_{k_2}(b_j) = T_*$.

Using a result from [5], it follows that the adversary's success probability is at most $2^{tq}\varepsilon\delta$.

Remark 4. *Note that the forgerer is allowed to choose the message and the tag (M_*, T_*) but not to change the nonce (b_j). This corresponds to the forgery that the MEE adversary is required to carry out at some level of the integrity tree (see Remark 2). The bound $2^{tq}\varepsilon\delta$ follows from adapting the proof of the main theorem of [5] to the limited forgery case. This is why we do not use here the maximum $(q+1)$-interpolation probability of f_{k_2}, as in [5] (where the forgery can also modify the nonce).*

4.3 The MEE encryption scheme and its confidentiality bound

The MEE encryption scheme is a “tweaked” AES Counter Mode, operating on the message space $\Omega = \{0,1\}^{512} \times \{0,1\}^{\alpha} \times \{0,1\}^{\beta}$, for parameters $\alpha, \beta > 0$ that satisfy $\alpha + \beta \leq 126$. A CL in the Protected data region is viewed as a triplets $(M, x, y) \in \Omega$, where $M \in \{0,1\}^{512}$ is the plaintext/ciphertex and $x \in \{0,1\}^{\alpha}$ and $y \in \{0,1\}^{\beta}$ are its spatial and temporal coordinates, respectively.

Definition 1 (MEE encryption scheme)**.** *Let* $(M, x, y) \in \Omega$ *be a triplet. View* $M = M[511:0]$ *as the concatenation* $U_3 \| U_2 \| U_1 \| U_0$ *of* 4 *128-bit blocks* $U_j = M[128 \cdot j + 127 : 128 \cdot j]$, $j = 0,1,2,3$. *For* $j = 0,1,2,3$, *write* j *in base* 2 *as a* 2*-bit string* $j_{[2]} \in \{0,1\}^2$. *Define the* 4 *compound nonces* $CTR_j \in \{0,1\}^{128}$ *by*

$$CTR_j = 0^{126-\alpha-\beta} \| x \| j_{[2]} \| y \tag{2}$$

and let

$$V_j = U_j \oplus AES_{K_{ENC}}(CTR_j) \tag{3}$$

The encryption of (M, x, y) *is* $E(M, x, y) = V_3 \| V_2 \| V_1 \| V_0 \in \{0,1\}^{512}$.

Decryption is analogous. In our implementation, $x = Addrs >> 6$, where $Addrs$ is the address of the first byte of the CL. The architecture has a 40 bits address space, implying $\alpha = 34$. The temporal coordinate y is the associated version and we choose $\beta = 56$. These parameters satisfy $\alpha + \beta \leq 126$. An illustration is shown in Figure 4 (left panel).

Suppose that an adversary submits q' queries to collect a valid trace T' of ciphertexts $E(M_i, x_i, y_i)$, $i = 1, \dots q'$, from chosen triplets (M_i, x_i, y_i). Suppose that $q' \leq 2^{56} - 2$ (by Property 4, the pairs (x_i, y_i) in T' are distinct). Then, we have the following confidentiality bound.

Proposition 1 (Confidentiality bound)**.** *Let* **Adv** *be the advantage of a probabilistic polynomial time algorithm in distinguishing the ciphertexts in* T' *from a set of random strings. Then,*

$$\mathbf{Adv} \leq \varepsilon_{AES}(q') + \frac{(q')^2}{2^{125}} \tag{4}$$

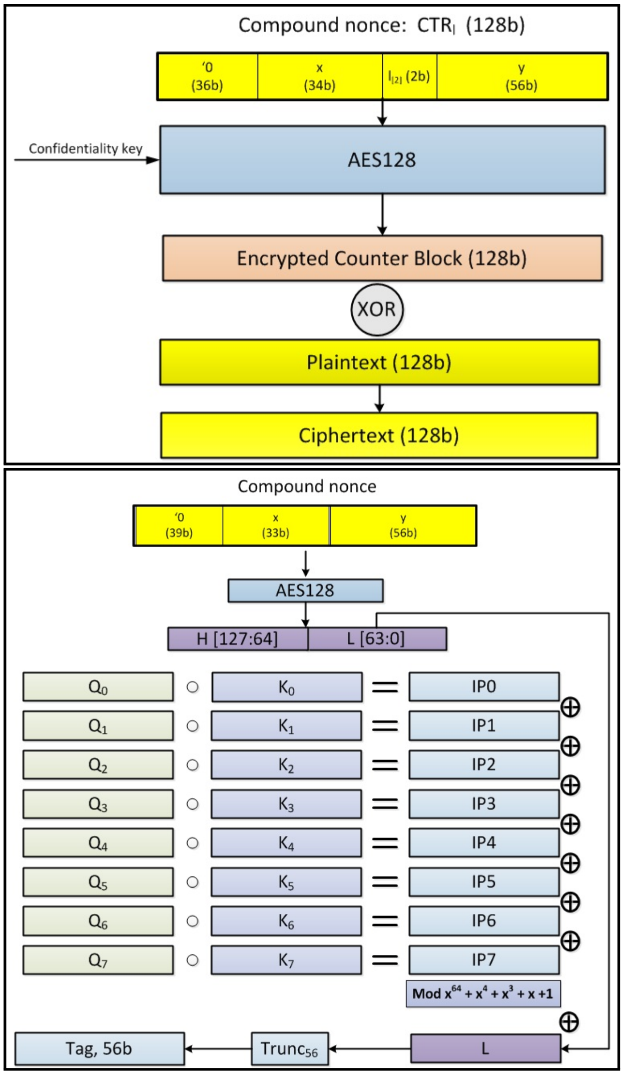

Figure 4: The MEE cryptographic primitives, implemented with the choice $\alpha = 34$, $\beta = 56$, $t = 56$. Left panel: the encryption. Right panel: the MAC algorithm.

Proof. With distinct pairs (x_i, y_i) $i = 1, \dots, q'$, the CTR_j values in Eq. (2) are also distinct. A CL includes 4 blocks, so each query translates to observing 4 ciphertext blocks. The bound in (4) follows from the standard birthday bound ($\frac{(4q')^2}{2^{129}}$ in our case) for distinguishing a random permutation from a random function, and from the definition of $\varepsilon_{AES}(q')$. □

4.4 The MAC algorithm and its integrity bound

The MEE Carter-Wegman MAC instantiation. Figure 4 (right panel) shows a schematic illustration of the MAC algorithm. We choose the parameters $d_1 = s_1 = 512$, $t = 56$, and $d_2 = s_2 = 128$, and specify H and f as follows.

$$f_{k_2}(b) = \text{Truncate}_t\left(AES(k_2, b)\right) \tag{5}$$

with $k_2 = K_{MAC}$, and where "Truncate$_t$" denotes truncation of a sting to its t least significant bits. The MAC algorithm takes an input triplet (M, x, y) and defines b as the compound nonce

$$b = 0^{128-\alpha-\beta}\ \|x\ \|y \tag{6}$$

(recall that $\alpha = 34$, $\beta = 56$; $\alpha + \beta \leq 128$). To define the family H for $X \in \mathsf{D}_1$, we view D_1 as $(\mathbb{F}_{2^{64}})^8$, and set

$$h_{k_1}(X) = \text{Truncate}_t\Bigg(\sum_{j=0}^{7} X[64\cdot j+63 : 64\cdot j] \otimes k_1[64\cdot j+63 : 64\cdot j]\Bigg) \tag{7}$$

Here, $\otimes$ is the multiplication in $\mathbb{F}_{2^{64}} = \mathbb{F}(2^{64})/(x^{64} + x^4 + x^3 + x + 1)$, and $X = X[511:0]$ is viewed as the concatenation $X_7\|X_6\|X_5\|X_4\|X_3\|X_2\|X_1\|X_0$ of the 8 field elements $X[64\cdot j+63 : 64\cdot j] \in \mathbb{F}_{2^{64}},\ j = 0,\ldots,7$. The hash key k_1 is $(K_7, K_6, \ldots, K_0)$. This (truncated) multilinear hash function is XOR-universal (i.e., $\varepsilon = 2^{-t} = 2^{-56}$).

Applying the MAC algorithm function to different CL types. The MEE uses (M, x, y) in different ways, according to the type of the CL. For Protected data CL's, x is its address, y is its version, and Eq. (7) is applied to $M[511:0]$. For a CL that holds versions or counters, x is the address and y is the associated counter on the integrity tree. To capture only the 8 versions/counters and skip the embedded MAC tags of such CL's (Section 3.2), Eq. (7) is applied to $X[511:0] = M[511:0]\&\big(Q\|Q\|Q\|Q\|Q\|Q\|Q\|Q\big)$ where $Q = 0^8||1^{56} \in \{0,1\}^{64}$. Note that with this definition, the compound nonces (Eq. (6)) do not repeat in an adversary-collected trace that has at most $2^{56} - 2$ samples.

The forgery resistance of the MAC algorithm. To bound the maximum interpolation probability of f, defined in Eq. (5), we approximate AES as a random permutation (of $\{0,1\}^{128}$) and find a bound on the maximum q-interpolation probability, denoted $MaxIntProb_t(q)$, of a *truncated* permutation to t bits. For any choice of q $(\leq 2^{64})$ values $a_j \in \{0,1\}^t$, $j = 1,2,\ldots,q$ (not necessarily distinct), the number of permutations P of $\{0,1\}^{128}$ such that the condition $P(b_j)[t-1:0] = a_j$, $j = 1,2,\ldots,q$ (for some distinct b_j), is at most $2^{(128-t)q}\cdot(2^{128}-q)!$ (equality holds only if all the a_j are distinct). It follows that

$$\begin{aligned} MaxIntProb_t(q) &\leq \frac{2^{(128-t)q}\cdot(2^{128}-q)!}{(2^{128})!} = \\ = 2^{-tq}&\cdot\frac{2^{128q}}{(2^{128})\cdot(2^{128}-1)\cdot(2^{128}-2)\cdot\ldots\cdot(2^{128}-q+1)} = \\ &= 2^{-tq}\cdot\frac{1}{(1)\cdot(1-\frac{1}{2^{128}})\cdot(1-\frac{2}{2^{128}})\cdot\ldots\cdot(1-\frac{q-1}{2^{128}})} \end{aligned} \tag{8}$$

We use the Bernoulli inequality

$$(1-a_1)\cdot(1-a_2)\cdot\ldots\cdot(1-a_k) \geq 1-(a_1+a_2+\ldots+a_k) \tag{9}$$

for any $a_j \geq 0$, $j = 1,2,\ldots,k$ such that $\sum_{k=1}^{k} a_j \leq 1$. Apply (9) to $a_j = \frac{j-1}{2^{128}}$. Note that $\frac{1}{1-w} \leq 1+2w$ for $0 < w \leq \frac{1}{2}$, and recall that $q \leq 2^t - 2 < 2^{64}$ by assumption. This gives

$$\begin{aligned} MaxIntProb(q) &\leq 2^{-tq}\cdot\frac{1}{1-\frac{(q-1)q}{2^{129}}} \leq \\ &\leq 2^{-tq}\cdot\frac{1}{1-\frac{q^2}{2^{129}}} \leq 2^{-tq}\cdot\left(1+\frac{q^2}{2^{128}}\right) \end{aligned} \tag{10}$$

We can now apply the result of [5], and conclude the following.

Proposition 2 (The MEE forgery resistance)**.** *An active adversary who collects a trace of $q \leq 2^{56} - 2$ message-tag samples that the MEE produces, and attempts a forgery, has success probability at most*

$$\begin{aligned} P_{success}(q) = \varepsilon_{AES}(q) + \varepsilon\cdot\left(1+\frac{q^2}{2^{128}}\right) &\leq \\ \leq\ \varepsilon_{AES}(2^{56}) + \frac{1}{2^{56}}\cdot\left(1+\frac{1}{2^{16}}\right) \end{aligned} \tag{11}$$

Remark 5. *The bound in (11) shows that the forgery success probability can be (theoretically) improved by only a marginal amount even if the adversary is able to collect the longest allowed trace. From the viewpoint of an attack strategy, it turns out that the trace collection time invested to increase the forgery success probability, has insignificant contribution. Therefore, a repeated "blind guessing", with success probability* $\frac{1}{2^{56}}$*, is the more efficient attack strategy in our context.*

5 Notes on MEE implementation and optimizations

Incrementing the counters. The hardware cost of implementing $INCREMENT(A)$ is lower than the cost of implementing a 56-bit adder (or counter). It requires only a 56-bit Galois Shift Register with taps in positions 34, 35, 55, where $INCREMENT(A) = (A << 1) \oplus (0x180000C00000001\&(-A[55]))$ ($A[55]$ is bit 55 of the string A).

The MAC tag computations in $\mathbb{F}_{2^{64}}$: Implementing the $\mathbb{F}_{2^{64}} = \mathbb{F}(2^{64})/(x^{64}+x^4+x^3+x+1)$ multiplication requires only one 64-bit (binary) polynomial multiplier. To compute the hash of a CL, the hardware accumulates (XOR's) 8 polynomial products into a 128-bit container, and then reduces modulo $(x^{64}+x^4+x^3+x+1)$. The reduction re-uses (twice) the same polynomial multiplier circuitry.

Truncating the MAC to 56 bits. It is possible to choose tags with $t = 64$ bit without changing the layout of the MEE data structure (i.e., a CL with 8 56-bit counters can embed a 64-bit tag). However, by truncating to $t = 56$ bits, the hardware can use the same data bus for counters, versions, and tags. This presents a useful area optimization. It is possible to view a CL as the concatenation of 10 elements in $\mathbb{F}_{2^{56}}$and apply a multilinear hash function over $\mathbb{F}_{2^{56}}$ without truncation. Our study showed that the area saving from this implementation is marginal, and does not justify the added complication that is involved is such design.

Efficient AES hardware implementation. A high-bandwidth low latency AES hardware unit is a critical component for supporting the multiple AES computations that are carried out during the MEE operations. On the other hand, integrating hardware units into the microprocessor die, where resources are very scarce, requires a low area solution. To address these conflicting requirements, the MEE uses an optimized AES design that is based on the most efficient representation of $GF(2^8)$ as $GF(2^4)^2$ [9].

The MEE cache A full walk over the MEE integrity tree involves multiple memory accesses. For example (see Figure 3), reading and verifying a CL requires the MEE to access six CL's from the DRAM (plus one access to the SRAM array). Writing (with tree update) requires more accesses. To alleviate the resulting performance loss, the MEE is equipped with an internal dedicated "MEE cache" that stores versions and counters from the tree (not data CL's or PD_TAG's). The hardware is configured to stop the tree walk when an access to such a value hits the internal cache. For example, a read operation with an MEE cache hit at the (Metadata) version level, requires the MEE to access only two CL's (the data and the PD_Tag) on the DRAM.

6 Real world analysis: challenging the cryptographic bounds

We challenge the cryptographic bounds of the 56-bit MAC tags and 56-bit counters, under an idealized adversary assumption (see Section 2). This adversary can harvest chosen input memory traces at absolute accuracy, with no time spent on copying and storing them for analysis. He is limited only by the throughput of the underlying hardware. For this adversary, we calculate whether (or not) it is practical to a) gain information on the plaintext from observing enough ciphertext samples; b) rollover a counter; c) make a successful forgery attempt.

Data collection and counter propagation rates. Collecting a trace T' (of ciphertexts) or T" (of message-tag pairs) with q samples, requires the MEE to write q Protected data CL's and update the integrity tree accordingly. In the fastest scenario, where all these writes enjoy a full

MEE cache hit, producing one sample involves writing two CL's (the data and the MAC tag CL's). Similarly, propagating a counter (version) requires the MEE to write two CL's. For each "write + tag computation", we have the following operations count: 5 AES operations (4 for encrypting the CL, and 1 for the MAC tag) and 10 polynomial multiplications (8 for the multilinear hash function and 2 for the reduction). Consequently, trace collection and counter propagation rates are bounded by the minimum between: a) $\frac{1}{5}$ the throughput of the AES engine; b) $\frac{1}{10}$ the throughput of the polynomial multiplier hardware; c) $\frac{1}{2}$ the throughput of the MC.

Bounding the rate of active attacks. Recall (Per Remark 5) that collecting information from a trace has such a marginal contribution that there is no benefit in investing the data collection time. Therefore, we assume that adversary's strategy is "blind guessing": resetting the system, loading the "victim" enclave, and attempting some (arbitrary) memory modification on a targeted memory location. A failure forces a reset with new keys and the attempt is repeated. With success probability $p_0 = \frac{1}{2^t} = 2^{-56}$, the expected number of attempts until the first success is $\sum_{k=0}^{\infty} k(1-p_0)^{k-1} = \frac{1}{p_0} = 2^{56}$. Let us assume that the attack is performed on a custom OS that bypasses all the BIOS overheads, and ignore the time to setup SGX, load (and run) the attacked code. Suppose also that the attack is automated, and some hardware unit physically resets the system when a MAC mismatch is identified, and leads to immediate power-off without waiting for the system to hang due to the drop-and-lock chain of events. Ignoring all these (and more), we estimate that pure setup time is still bounded by, say, 1ms.

System capabilities. The throughput of the AES unit in the MEE is 1 AES block per cycle, and the throughput of the polynomial multiplier is 1 multiplication per cycle. The throughput of the MC is $32 \cdot 2^{30}$ bytes/sec (32GB/sec). All these units operate at a frequency that is limited to 3.2GHz (under overclocking).

Translating to lower bounds on the attack time.

1. Collecting 2^{56} ciphertext samples (to get distinguishing advantage at most 2^{-13}), or, equivalently, rolling over a counter, is a serial process that takes at least $\sim$ 8 years of continuous processing on a dedicated platform.
2. The expected time to make a successful forgery is $\sim$ 2M years on a customized platform (note that this task can be distributed over multiple platforms).

We conclude that the cryptographic security margins of the MEE design are sufficient even under adversarial assumptions that are way beyond what can be considered practical capabilities.

7 Performance study

Estimating the performance cost of the MEE is a delicate task, because the results depend heavily on the application that is measured, and on isolating the MEE effect from other SGX overheads. We report here some preliminary performance results from a repeatable experiment that we carried out.

The experiment used the 445.gobmk component of SPECINT2006 v01 [2], selecting 10 input files (namely: arb.tst, arend.tst, blunder.tst, trevorc.tst, nicklas4.tst, nicklas2.tst, nngs.tst, buzco.tst, atari_atari.tst, score2.tst). We compiled the 445.gobmk test with Graphene (library OS [1]), after adapting it to run inside an Intel SGX enclave. This test was subsequently measured with the 10 input files, under two conditions: a) without SGX (and hence no MEE involved); b) inside an enclave (i.e., while the MEE is active).

Our goal was to try to isolate, as much as possible, the effect of the MEE. To this end, we selected a test/enclave that would incur almost no other SGX overhead, aside from the MEE overhead itself. The enclave was entirely loaded into the MEE memory region, had no page swapping across the protected region and the general purpose DRAM, had only a few transition to/from enclave mode, and no I/O. We believe that the results of this experiment provide a good approximation of the MEE overhead. The measurements were taken on an engineering sample of the processor (Architecture Codename Skylake). The results are shown in Figure 5. We see that the MEE imposes performance degradation that varies from 2.2% to 14%, with an average of 5.5%.

Note that we expect to see variations in the performance results, that depend on the intensiveness of the memory usage of an application. Indeed, in our test, the different input files induce different memory utilization patterns that change the number of CPU cache misses (thus also the internal MEE cache misses when running from an enclave). To illustrate, we give one extreme example. Our profiling shows that gobmk test with score2.tst input has 979M cache misses, whereas the same test with nngs.tst has only 549M cache misses. Indeed (See Fig. 5), the overall performance reflects this different behavior. We also point out that different input files (e.g., blunder.tst) may trigger different code flows of the test.

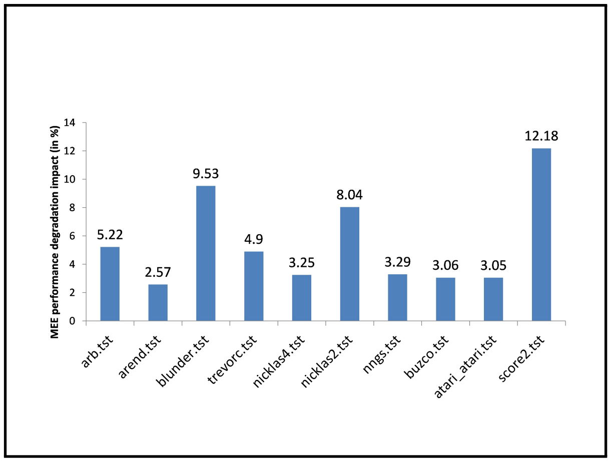

Figure 5: Performance comparison of the 445.gobmk component of SPECINT 2006, with 10 input files (see explanations in the text). The bars show that the performance degradation (in %) incurred by enabling the MEE, varies from 2.2% to 14%, with an average of 5.5%.

8 Conclusion

This paper detailed the design of an MEE instantiation that is implementable under the strict engineering constrains of a general purpose microprocessor. It enjoys proven cryptographic properties which we analyzed under an idealized adversarial model, and challenged in a real-world context.

The efficiency of this design is demonstrated by its being part of the latest 6th Generation Intel Core processor[3] as a critical hardware component that supports the SGX technology.

The design is flexible and can be tweaked to accommodate different targets. We give two possible tradeoff examples: a) The data units of the Protected data region can be extended to cover several CL's, thus associating a PR_Tag to more than 512 bits. This will trade a smaller seized region on the DRAM with some performance degradation; b) The MEE can be built to use multiple encryption (and MAC) keys in order to extend the effective length of the counters.

Acknowledgments

We thank Ittai Anati, Ilya Alexandrovich, Ernie Brickell, Dan Boneh, Jack Doweck, Kevin Gotze, Rafael Misoczki, Benny Pinkas, Ronny Ronen, and Jesse Walker, for useful comments. S. Gueron was partly supported by the Blavatnik Interdisciplinary Cyber Research Center (ICRC) at the Tel Aviv University.

References

[1] –. Graphene. `http://graphene.cs.stonybrook.edu/`. Accessed: 2015-09-30.

[2] –. Specint. `https://www.spec.org/cpu2006/CINT2006/`. Accessed: 2015-09-30.

[3] –. Intel ® 64 and IA-32 architectures software developer's manual. *Volume 3a: System Programming Guide* (September 2015).

[4] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), vol. 13 of *HASP '13*.

[5] Bernstein, D. J. Stronger security bounds for Wegman-Carter-Shoup authenticators. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings* (2005), pp. 164–180.

[6] Duc, G., and Keryell, R. Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual* (Dec 2006), pp. 483–492.

[3] Formally, 6th Generation Intel® Core™ processor

[7] Elbaz, R., Champagne, D., Gebotys, C., Lee, R. B., Potlapally, N., and Torres, L. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Transactions on Computational Science IV, Lecture Notes in Computer Science (LNCS)* (March 2009 2009), 1–22.

[8] Gassend, B., Suh, G. E., Clarke, D., van Dijk, M., and Devadas, S. Caches and merkle trees for efficient memory authentication, 2002.

[9] Gueron, S., and Mathew, S. Hardware implementation of AES using area-optimal polynomials for composite-field representation $GF(2^4)^2$ of $GF(2^8)$. Manuscript, 2016.

[10] Henson, M., and Taylor, S. Beyond full disk encryption: Protection on security-enhanced commodity processors. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security* (Berlin, Heidelberg, 2013), ACNS'13, Springer-Verlag, pp. 307–321.

[11] Henson, M., and Taylor, S. Memory encryption: A survey of existing techniques. *ACM Comput. Surv. 46*, 4 (Mar. 2014), 53:1–53:26.

[12] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 11:1–11:1.

[13] Hou, F., He, H., Xiao, N., Liu, F., and Zhong, G. Efficient encryption-authentication of shared bus-memory in SMP system. In *10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010* (2010), pp. 871–876.

[14] Hu, Y., Hammouri, G., and Sunar, B. A fast real-time memory authentication protocol. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing* (New York, NY, USA, 2008), STC '08, ACM, pp. 31–40.

[15] institute˜of˜standards˜and technology, N. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) Family. Tech. rep., DEPARTMENT OF COMMERCE, Nov. 2007.

[16] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP '13, ACM, pp. 10:1–10:1.

[17] Nagarajan, V., Gupta, R., and Krishnaswamy, A. Compiler-assisted memory encryption for embedded processors. In *Proceedings of the 2Nd International Conference on High Performance Embedded Architectures and Compilers* (Berlin, Heidelberg, 2007), HiPEAC'07, Springer-Verlag, pp. 7–22.

[18] Nechvatal, J., Barker, E., Bassham, L., Burr, W., Dworkin, M., Foti, J., and Roback, E. Report on the development of the Advanced Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology 106* (2001). URL: `http://nvl.nist.gov/pub/nistpubs/jres/106/3/cnt106-3.htm`.

[19] Rogers, B., Chhabra, S., Prvulovic, M., and Solihin, Y. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2007), MICRO 40, IEEE Computer Society, pp. 183–196.

[20] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing* (New York, NY, USA, 2003), ICS '03, ACM, pp. 160–171.

[21] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing* (New York, NY, USA, 2003), ICS '03, ACM, pp. 160–171.

[22] Thekkath, D. L. C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural support for copy and tamper resistant software. *SIGARCH Comput. Archit. News 28*, 5 (Nov. 2000), 168–177.

[23] Wegman, M. N., and Carter, L. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci. 22*, 3 (1981), 265–279.

[24] Williams, D., and Sirer, E. G. Optimal parameter selection for efficient memory integrity verification using merkle hash trees. In *Network Computing and Applications, 2004. (NCA 2004). Proceedings. Third IEEE International Symposium on* (Aug 2004), pp. 383–388.

[25] Yan, C., Englender, D., Prvulovic, M., Rogers, B., and Solihin, Y. Improving cost, performance, and security of memory encryption and authentication. *SIGARCH Comput. Archit. News 34*, 2 (May 2006), 179–190.

[26] Zhuang, X., Zhang, T., and Pande, S. Hide: An infrastructure for efficiently protecting information leakage on the address bus. *SIGOPS Oper. Syst. Rev. 38*, 5 (Oct. 2004), 72–84.