Memory Encryption for General-Purpose Processors

Shay Gueron | Intel Corporation and University of Haifa

Intel's Memory Encryption Engine represents a successful feat of real-world cryptographic engineering: it's the first time that cryptographic memory protection has been added to a widely deployed general-purpose processor.

Intel's Software Guard Extensions (SGX) is a security technology that allows general-purpose computing platforms to run software in a trustworthy manner and handle secrets. It was introduced in the 6th Generation Intel Core Processor.

The SGX trust boundary includes only the CPU internals. Its vulnerable external memory (DRAM) must be cryptographically protected. The required protection is provided by the Memory Encryption Engine (MEE), a new hardware unit that's been added to the processor's memory controller. It protects the confidentiality, integrity, and freshness of the CPU-DRAM traffic against eavesdropping and tampering.

The MEE is a successful feat of real-world cryptographic engineering: it's the first time such cryptographic memory protection has been added to a widely deployed general-purpose processor. In this article, I explain the MEE threat model and security objectives, describe some of the difficulties encountered in building the MEE, and report concrete performance results.

Anything on Your System Memory Can and Will Be Used against You in an Attack

Attacks on a computing device's DRAM pose serious security threats for the mobile market, where attackers can gain physical access to devices through theft or loss, as well as for cloud computing environments, where attackers can gain access to applications that use customers' private data in an uncontrolled and, hence, untrusted facility.

All it takes to launch a memory attack is physical access to a platform that allows attackers to connect tools for reading and modifying the DRAM. It has been demonstrated that interfaces such as FireWire and Thunderbolt can be used to read and write physical memory.[1,2] Such tools are readily available and serve legitimate purposes: commercial entities and researchers have been using memory primitives for remote debugging,[3] memory capture for forensics purposes,[4] and legal interception.[5] Therefore, methods to protect memory by limiting attackers' abilities to access, control, or tamper with it isn't always enforceable or sufficiently secure.

Many different kinds of attacks have been demonstrated, from compromising secrets that reside on the DRAM using a "cold boot attack"[6] to bypassing policies and security mechanisms,[7–9] installing malware,[10] and running an entire exploit payload.[11]

Memory Attackers

I consider two types of attackers:

 Copublished by the IEEE Computer and Reliability Societies

- *Passive eavesdroppers* who can read the DRAM contents. Their goal is to collect the confidential information that resides there.
- *Active forgers* who can read and modify the DRAM contents. The modified contents are eventually read back from the DRAM to the CPU and then used. The goal is to change an application's or system's behavior and exploit the emerging vulnerability. A replay attack is a particular form of forgery that involves overwriting the DRAM with a previous value.

What Do Attackers Seek?

Memory attacks can be carried out against different targets and in different scenarios. We follow the Branco-Gueron A-B-C categories:[12]

- *Access-seeking attackers.* These attackers don't own the platform, but gain possession of it in a locked state. They want to obtain user access to steal system data.
- *Breaching attackers.* These attackers are legitimate platform users who want to breach some of the system's policies or circumvent restrictions on their privileges.
- *Conspirator attackers.* These attackers are also legitimate users of the platform or environment. They have administrative powers and conspire to collect other users' data.

Potential scenarios for each of the A-B-C attackers include a stolen device for A, a corporate employee for B, and an employee in a cloud provider's facility for C.

Cryptographic Memory Protection

Cryptographic memory protection's goals are to maintain data confidentiality and guarantee data integrity and freshness by preventing forgery and replay. Some scenarios can be settled with encryption only. One example is the cold boot attack[6] that's executed by physically removing the DRAM, thus harvesting a (single) snapshot of its contents. The aim is to read the information from that snapshot.

This attack can be mitigated by encrypting the memory, which will deliver to the cold boot attacker a sample of worthless ciphertext.

Encryption Alone Isn't Enough!

Encryption protects the confidentiality of data. It seems reasonable to assume that encrypting memory will also, as a byproduct, protect against active attackers, because encryption limits active attackers to blinded random block corruption attacks. In these attacks, some unknown value (ciphertext) on the memory is modified with the hope of achieving an attack goal when the CPU ends up using a randomly corrupted block of (decrypted) data. Could this be a "good enough" approach? Reality suggests it isn't.

Figure 1. The Branco-Gueron blinded random block corruption attack demonstration, using the XDP/J-TAG (eXtended DebugPort/Joint Test Action Group) interface.[12] The XDP device (blue box) is connected through the J-TAG port of the CPU on the attacked machine (the board). The black screen shows the login prompt on the victim machine. The memory write randomly corrupts the DRAM contents and causes a change in the login process execution flow. The attacker logs in a "root."

Figure 1 demonstrates how an attacker can become a root on a locked system even if the memory is encrypted, and the only available interaction with the device is the login screen that requests a password, as long as a memory read/write interface remains available to the attacker.

SGX in a Nutshell

SGX technology is complex, involving many details.[13–17] This article describes it briefly, to the extent necessary to illustrate SGX's need for memory protection.

SGX trusts only the CPU internals. The assumed adversary has full control of the system and the software running on it at any privilege level, and can read and modify the DRAM contents (including copy-and-replay). SGX consists of a set of CPU instructions and is supported by an internal hardware-based access control mechanism. The instructions provide a way to load application code and data from memory while incrementally locking it in a dedicated DRAM region, and trustworthily generating its cryptographic identity (SHA-256 digest). Once the code is loaded, it can run as a secure "enclave," remaining isolated (via access control) from all other system processes. Because application code is merely an observable (and thus auditable) binary file, it can't ship with secrets. Consequently, SGX lets a secret owner provision a

secret to a trustworthy enclave. Instructions, together with an attestation protocol and infrastructure, provide tools that let an enclave prove to an off-platform entity that it's running on a genuine processor under the SGX restrictions and that the cryptographic identity it reports is trustworthy. With this, a secret owner can establish a secure channel with the trustworthy (and preaudited) enclave, and provision a secret. To handle these secrets, SGX has instructions that a running enclave can invoke and obtain a secret key that's unique to the platform and its identity. The enclave can use this key to encrypt information to an untrusted location (disk) and decrypt it in subsequent runs.

Clearly, SGX requires full memory protection for the memory range that enclaves (and the SGX firmware) use. Handling secrets during runtime requires CPU-DRAM traffic to be confidential. Trusting that an enclave executes its intended flow and that its reported cryptographic identity is legitimate depends on tamper-resistant DRAM. Note that replay prevention is critical. Without it, attackers can take a piece of code (and the matching integrity values) from the DRAM, while a legitimate enclave is running, and replace it with a recorded image from a previously running fake enclave. The modified enclave will continue running with a changed behavior while the CPU reports the measured identity of the legitimate enclave.

MEE Security Objectives and Adversarial Model

The MEE is designed to

- protect the confidentiality of the data written to the DRAM; and
- protect data integrity and prevent replay—ensuring that the data the MEE reads back from the DRAM is the same data most recently written by the MEE to the DRAM.

Adversaries

Both passive eavesdroppers and active forgers can read DRAM content, but only active adversaries can modify it. Adversaries collect information by observing and recording DRAM snapshots for a set period of time. The snapshots consist of ciphertext, metadata, and tree values, which are produced by the activities of victim enclaves that write secret data, benign enclaves that write known data, and crafted attack enclaves that write adversary-chosen data.

Passive adversaries target a secret that's encrypted to the DRAM by a victim enclave. They extract q' samples of encrypted cache lines (CLs) from the observed snapshots. Subsequently, they apply any algorithm to acquire information on a targeted secret. The confidentiality promise of the MEE encryption is assessed by an upper bound on the adversaries' advantage in distinguishing MEE-produced ciphertext from a random output, after observing q' samples.

Active adversaries collect q'' samples. Subsequently, they apply an algorithm and, at some chosen time, modify the DRAM contents in any chosen way. The attack succeeds if the modified data is read back and passes the MEE integrity check. A replay attack is a particular form of forgery carried out by overwriting the DRAM with a previous value. The MEE policy is to tolerate only one failing forgery attempt with a given set of keys (it crashes the system when the integrity checks fail). However, adversaries can repeat the attack after each system crash with newly generated keys. In a degenerate strategy, the forgers spend no time at all observing samples, proceeding directly to blind-guess forgeries. The MEE integrity promise is assessed by an upper bound on the probability to succeed in the first forgery attempt, after observing q'' samples, and the maximum rate for repeating such attempts.

Real-World Analysis Using an Idealized Adversary

It's important to determine a strategy for assessing MEE's security promise as part of its design. The strategy will depend on the assumptions made about the adversaries' capabilities.

Our strategy is to leave the widest possible security margins. Astute adversaries will try to generate the most favorable (for their attack) set of samples within a given time frame. Their actual ability to do so depends on many practical factors. Our analysis strategy is to avoid any assumptions about the adversaries' capabilities. We therefore assess the MEE against an imaginary idealized adversary who can record (and modify) memory snapshots with absolute accuracy at the granularity of atomic CL read/write operations, spends no time on copying and storing the data (for offline analysis), and receives all the q' (q'') samples from his or her chosen input. This adversary is limited only by the physical capacity of the MEE hardware to produce samples—by far beyond anything feasible in reality.

Under the idealized adversary model, analysis of the MEE's strength depends on setting lower bounds on the time required to

- produce q' (q'') samples,
- repeat forgery attempts until the first success, and
- execute enough writes that propagate the MEE state to its limit; that is, the largest number of samples q' (q'') that the system can possibly produce before it's "exhausted" and restarts the system with new keys.

Designing the MEE

Intensive studies of the memory protection problem have led to various elegant solutions. However, these studies were mostly based on simulations and field-programmable gate array prototypes and weren't integrated into a real general-purpose processor. Indeed, the long-standing memory protection problem is surprisingly difficult to address in practice.

Like most memory encryption technologies, our design is based on two pillars:

- the cryptographic primitives that realize encryption, message authentication, and anti-replay mechanisms; and
- an integrity tree that protects a large memory region with a smaller user-inaccessible region as well as very small on-die storage.

The difference is in the details, where innovation is extremely useful: the lean resources of real systems and the high complexity of integrating an autonomous hardware unit in the processor mandate aggressive optimization.

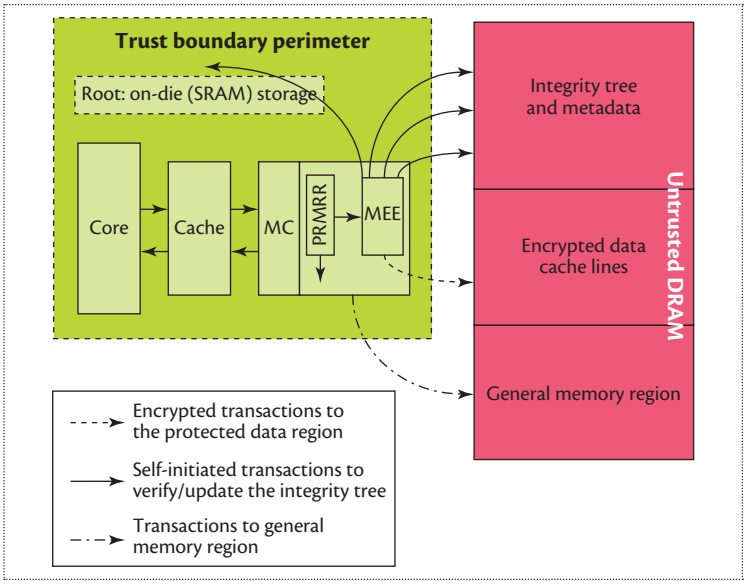

Figure 2. Schematic illustration of the Memory Encryption Engine (MEE) operation. The MEE hardware unit functions as an extension of the memory controller (MC), which routes read/write requests to the relevant MEE region. The MEE encrypts/decrypts the data before sending/fetching it to/from the DRAM. PRMRR is processor reserved memory range registers.

Challenges in Adding MEE to General-Purpose Processors

To give readers a taste of how difficult designing the MEE was, I'll mention a few priorities, constraints, and requirements. Security is the ultimate goal, with no compromise allowed. In this framework, the optimizations attempt to reduce the internal storage, hardware area, performance penalties, user-inaccessible memory region, and overall complexity.

However, these targets are sometimes conflicting. Saving the tree size motivates using short authentication tags. Therefore, standard message authentication code (MAC) algorithms that produce long tags that need truncation are unattractive from a hardware cost viewpoint. Furthermore, tag truncation can deteriorate the security margins (for example, the standard AES-GCM [Galois Counter Mode]). Good performance benefits significantly from a parallelizable design.

Therefore, serial cipher block chaining encryption is undesirable. Extremely expensive on-die storage motivates use of integrity trees with many levels. However, these conflict with the desire to reduce user-inaccessible memory space and performance penalties. Adding dependencies between different components in the processor is expensive and complex, so a fully autonomous MEE is desired. However, this limits the possible MEE policies when integrity checks fail.

So, the challenge was to find a point in the design space that made the implementation sufficiently practical to become part of a real general-purpose processor.

System Overview and Basic Operations

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, the processor's core continuously issues memory transactions. All memory transactions are defined at the granularity of a 512-bit CL and are associated with a physical address in the memory space. Transactions that miss the cache are handled by the memory controller (MC) that reads/writes each CL from/to its address as an atomic operation.

SGX is enabled at boot time by a sequence of steps executed by microcode and a trusted firmware module. One step defines a range of addresses and programs it into the MC hardware (default size is 128 Mbytes). This range is accessible only to SGX firmware and running enclaves. The access control mechanism controls all accesses to this region (which is invisible to the OS).

The MEE hardware unit functions as an extension of the MC, which routes read/write requests to the relevant MEE region. The MEE encrypts/decrypts the data before sending/fetching it to/from the DRAM. In addition, it autonomously initiates additional transactions to handle metadata and verify or update the integrity tree. Figure 2 is a schematic illustration of how the MEE operates.

MEE keys. The MEE key material is generated by the trusted firmware module at boot time. It samples entropy from an on-package digital random number generator,

uniformly and randomly generates the required key bits, and programs them into the hardware-protected MEE registers. These bits are used for a 128-bit encryption key, a 128-bit authentication key, and a 512-bit hash key (used for the authentication algorithm).

Drop-and-lock policy. If the MEE detects a mismatch between fetched data and the expected authentication value, it emits a failure signal, drops the transaction (no unverified data ever reaches the cache), and immediately locks the MC (no further transactions are serviced). This eventually causes the system to hang and require resetting. The reset flow generates new MEE keys.

MEE Integrity Tree

An integrity tree is the standard method for protecting data with a small amount of internal storage. Many designs exist (good reviews of integrity trees are available in the literature[18,19]).

The classic method is the Merkle Tree (MT), a binary tree in which 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. A memory encryption technology can leverage the existence of an internal integrity key and gain efficiency by using a stateful MAC instead of a hash. A stateful MAC produces an authentication tag as a function of the data that has undergone MAC and a nonce (for example, a counter). Figure 3 shows the general concepts with the simplified examples. The top panel illustrates the simplistic approach: the on-die counters are trusted and provide an integrity guarantee for the MAC algorithm. Nonrepeating counters provide replay protection. Unfortunately, the cost of one counter per CL is prohibitive.

An integrity tree with several levels reduces the required internal storage. But it comes at the expense of confiscating user-inaccessible memory, and multiple memory accesses for read/write operations. Figure 3 (bottom) shows a tree with three levels, using a 4-ary layout. The 16 level-0 counters are protected by four MAC tags and four level-1 counters, which are protected by one MAC tag and an on-die level-2 counter (root). If these counters are properly maintained and used, the tree can protect the integrity and freshness of 16 data CLs.

Optimized MEE integrity tree. The cryptographic design applies an encrypt-then-MAC paradigm, where the MAC algorithm uses a nonce that combines the CL address and an associated counter. Encrypting a data CL produces its ciphertext and uses or updates the metadata: a MAC tag and a "version" counter. The version reflects the number of times the CL was written to the DRAM, information that's critical for achieving ciphertexts' semantic security over time. The version counters are protected by a tree with four levels and an 8-ary layout with level-3 counters stored on die.

Optimization is achieved by defining all counters and tags as 56 bits only. This allows the same CL to accommodate eight counters together with their tags, saving memory for the storing of tags in separate lines. This construction achieves a 4× compression factor between the data and metadata (tags and versions) regions, and an 8× factor between tree-level counters.

How much memory is left for the user? So how much internal storage does this cost? Altogether, 75 percent of the MEE region is user-available; the rest is overhead. With a default MEE region of 128 Mbytes, the construction avails 96 Mbytes for data CLs that applications can use. The root of the tree consumes only 4 Kbytes of internal storage.

MEE cache. A full walk over the MEE integrity tree involves accessing memory multiple times. For example, reading and verifying a data CL requires the MEE to access six CLs from the DRAM (plus one from on-die static RAM). Writing and updating the tree requires more accesses. To alleviate the performance penalty, the MEE is equipped with a dedicated internal "MEE cache" with fast access. It stores versions, counters, and data tags (not data lines). The hardware is configured to stop the tree walk when an access hits the internal cache—for example, a read operation that hits the MEE cache at the version and the MAC level requires only one DRAM access, which fetches the data CL.

MEE cryptographic functions. The MEE cryptographic design uses an encrypt-then-MAC approach. Encryption is implemented using the AES block cipher and a variation of counter mode (with nonce). The MAC algorithm is based on universal hashing. These primitives, which are used with a tree data structure and flows that combine memory address and a per-CL counter into a nonce, ensure that all encrypted blocks are distinct.

Encryption operates on a 512-bit CL, viewed as four 128-bit blocks. The critical property securing the encryption scheme is the guarantee that each block is encrypted with a counter block that's distinct across all addresses and times. This spatial and temporal uniqueness is achieved by constructing the counter blocks as a combination of the CL's address (34 bits), two bits that encode the location (0 to 3) of the block inside the CL, and the associated counter (56 bits).

The authentication scheme uses a Carter-Wegman MAC construction,[20] with a function from a universal

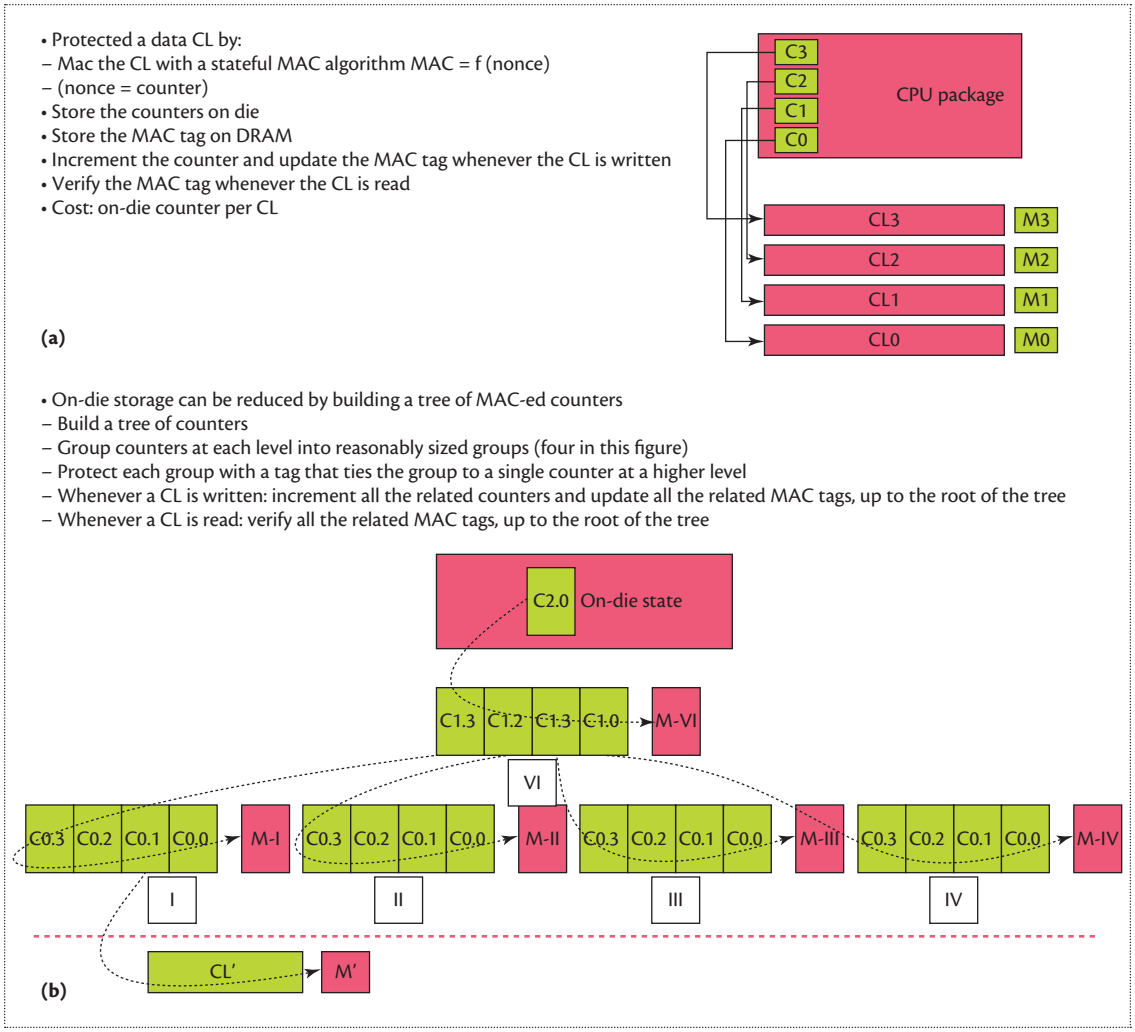

Figure 3. Integrity tree. (a) An on-die counter for each cache line (CL) is an approach to guaranteeing the integrity of a message authentication code (MAC) algorithm, but is cost prohibitive. (b) A three-level 4-ary integrity tree reduces the required internal storage. If the counters are properly maintained and used, the tree can protect the integrity and freshness of 16 data CLs.

family of multilinear hash functions selected by the hash key. Again, the construction and operations maintain the spatial and temporal uniqueness property.

Real-World Security Analysis of MEE Cryptographic Bounds

I'll now discuss how the MEE's level of security can be assessed based on proven cryptographic bounds. I'll then show how to check the feasibility of making an attack under the assumption that there are no limits on the adversary's capabilities, beyond the actual limitation of the underlying MEE hardware.

Cryptographic Bounds

In "A Memory Encryption Engine Suitable for General Purpose Processors," I state and prove the MEE construction's cryptographic bounds as a function of the number of samples (q'/q'') that the (passive/active) adversary collected.[21] Due to the counters' lengths, the possible number of samples with the same keys is no more than $2^{56} - 2$.

These bounds can be interpreted as follows:

- Even after seeing 2^{56} ciphertext samples, the adversary can't distinguish them from random outputs with probability greater than 2^{-13}.
- The probability of a forgery's success can be (theoretically) improved only by a negligible amount if the adversary collects approximately 2^{56} samples. This implies that the best attack strategy is repeated blind guessing, whose success probability per attempt is 2^{-56}.

Because our idealized adversary is limited only by the throughput of the underlying hardware, we can now check whether it's practical to gain information on some plaintext by observing a large amount of samples, exhaust the integrity tree, or make a successful forgery.

Data Collection and Counter Propagation Rates

Collecting q′ message-tag-nonce triplets requires the MEE to write q encrypted CLs and update the integrity tree accordingly. In the fastest possible scenario, all these writes enjoy a full MEE cache hit, so producing one sample only involves writing the data CL to memory, and the associated tag to the cache. This requires the MEE hardware to execute five AES operations and 10 polynomial multiplications. Consequently, the information collection and counter propagation rates are at most one-fifth of the AES hardware throughput, one-tenth of the polynomial multiplier hardware throughput, and one-half of the MC throughput.

The actual hardware capabilities are as follows:

- AES unit throughput in the MEE is one AES block per cycle,
- polynomial multiplier throughput is one multiplication per cycle, and
- MC throughput is 32×2^{30} bytes/s (32 Gbytes/s).

All these hardware units operate at a frequency that's limited to 3.2 GHz (under overclocking).

Repeated Active Attack Rates

A single blind-guess attack has success probability of 2^{-56}, so the expected number of attempts until the first success is 2^{56}. To estimate the highest possible rate of repeated forgery attacks, we assume, again, an idealized adversary. Assume that the attack is performed on a custom OS that bypasses all the basic I/O system overheads and ignore the time to set up SGX, load, and run the attacked enclave.

Suppose also that the attack is automated and that some hardware unit physically resets the system when a MAC mismatch is identified by immediately shutting the power off and not waiting for the system to hang due to the drop-and-lock chain of events. Ignoring all these events (and many more), we can bound the bare-bones setup time by an (unrealistic) 1 ms; that is, at most 1,000 forgery attempts per second are possible.

How Long Would an MEE Attack Take?

If we translate the cryptographic bounds and the hardware capability into a bound on the time that it would take an idealized adversary using an idealized machine to attack the system, we end up with these statements:

- Collecting 2^{56} ciphertext samples (for a distinguishing advantage of at most 2^{-13}) or, equivalently, rolling over a counter is a serial process. It takes at least eight years of continuous processing on a dedicated platform.
- The expected time to make a successful forgery is approximately two million years on the imaginary customized platform. This attack, however, can be distributed over multiple platforms.

Based on this real-world analysis, MEE offers adequate cryptographic security—even under adversarial assumptions that are way beyond anything practical.

Performance Cost of Adding MEE

Estimating the MEE's performance cost is a delicate task because the results depend on the measured application and on the ability to isolate the MEE impact from other SGX overheads. I report here the results of one experiment that used the 445.gobmk component of SPECINT2006 v01, selecting 10 input files (arb.tst, arend.tst, blunder.tst, trevorc.tst, nicklas4.tst, nicklas2.tst, nngs.tst, buzco.tst, atari atari.tst, and score2.tst).[22] The 445.gobmk test was compiled with Graphene (a library OS)[23] after adapting it to run inside an SGX enclave. This test was measured with the 10 files under two conditions: without SGX (no MEE involved) and inside an enclave (activated MEE).

Because our goal was to isolate MEE's effect as much as possible, we selected a test/enclave that incurs almost no other SGX overhead, aside from the MEE overhead. The enclave was entirely loaded into the MEE region, had no page swapping between the protected region and the general-purpose DRAM, had only a few transitions to and from enclave mode, and had no I/O. The measurements were taken on an engineering sample of the Skylake processor. The results, summarized in Figure 4, show that the MEE degraded performance between 2.2 and 12.2 percent, with an average of only 5.5 percent.

Note that variability in performance penalties is expected because performance depends on the intensiveness of the application memory usage. In our test, different input files induced different memory utilization patterns that changed the number of CPU cache misses (and, thus, also the internal MEE cache misses when running from an enclave). Consider one extreme example from our profiling: the gobmk test with score2.tst input had 979 million cache misses, whereas the same test with nngs.tst had only 549 million cache misses. Also note that different input files (for example, blunder.tst) might have triggered different code flows of the test. The results in Figure 4 reflect these different behaviors.

I reviewed, briefly, MEE's design highlights, with the goal of demonstrating some of the challenges involved in its building and how we analyzed and optimized it for final inclusion in an Intel general-purpose processor. Many technical details were omitted here for brevity, but I give a full account of MEE's construction in "A Memory Encryption Engine Suitable for General Purpose Processors."[21] ■

Acknowledgments

The Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University; the PQCRYPTO project, which is partially funded by the European Commission Horizon 2020 research programme (grant 645622); and the Israel Science Foundation (grant 1018/16) supported this research.

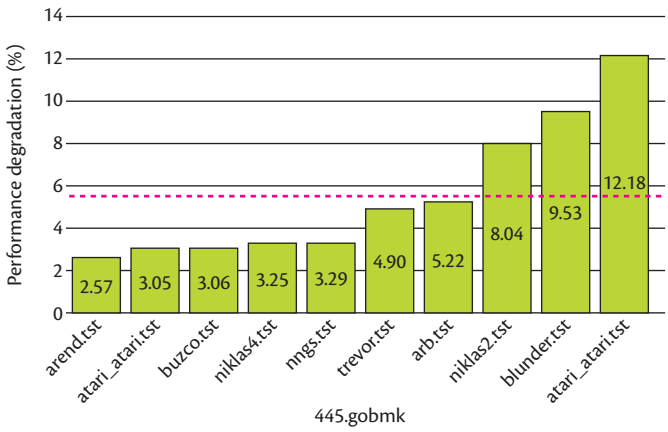

Figure 4. MEE performance study on the 445.gobmk component of SPECINT 2006, with 10 input files. The bars show that the performance degradation per input file that was incurred by enabling MEE varied between 2.2 and 12.2 percent, with the dashed horizontal line showing the average of 5.5 percent.

References

1. M. Becher, M. Dornseif, and C.N. Klein, "FireWire: All Your Memory Are Belong to Us," *Proc. CanSecWest Security Conf.*, 2005; cansecwest.com/core05/2005-firewire-cansecwest.pdf.
2. R. Sevinsky, "Adventures in Thunderbolt DMA Attacks," *Proc. Black Hat USA*, 2013; media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf.
3. J. Stecklina, "Remote Debugging via FireWire," diploma thesis, Technische Universitat Dresden, 2009; os.inf.tu-dresden.de/~jsteckli/diplom/diplom.pdf.
4. B.D. Carrier and J. Grand, "A Hardware-Based Memory Acquisition Procedure for Digital Investigations," *Digital Investigation: Int'l J. Digital Forensics & Incident Response*, vol. 1, no. 1, 2004, pp. 50–60.
5. N. Perlroth, "Software Meant to Fight Crime Is Used to Spy on Dissidents," *New York Times*, 30 Aug. 2012; www.nytimes.com/2012/08/31/technology/finspy-software-is-tracking-political-dissidents.html?_r=0.
6. J.A. Halderman et al., "Lest We Remember: Cold Boot Attacks on Encryption Keys," *Comm. ACM: Security in the Browser*, vol. 52, no. 5, 2009, pp. 91–98.
7. D. Aumaitre and C. Devine, "Subverting Windows 7 x64 Kernel with DMA Attacks," *Proc. Hack in the Box Security Conf.* (HITBSecConf 10), 2010; esec-lab.sogeti.com/static/publications/10-hitbamsterdam-dmaattacks.pdf.
8. L. Duflot, Y. Perez, and B. Morin, "What If You Can't Trust Your Network Card?," *Proc. 14th Int'l Conf. Recent Advances in Intrusion Detection* (RAID 11), 2011, pp. 378–397.
9. F.L. Sang, V. Nicomette, and Y. Deswarte, *I/O Attacks in Intel-PC Architectures and Countermeasures*, report, SysSec Consortium, 6 July 2011; www.syssec-project.eu/m/page-media/23/syssec2011-s1.4-sang.pdf.
10. P. Stewin and I. Bystrov, "Understanding DMA Malware," *Proc. 9th Int'l Conf. Detection of Intrusions and Malware, and Vulnerability Assessment* (DIMVA 12), 2013, pp. 21–41.
11. R. Breuk and A. Spruyt, *Integrating DMA Attacks in Exploitation Frameworks*, report, System and Network Engineering Lab, Univ. Amsterdam, 20 Feb. 2012; www.delaat.net/rp/2011-2012/p14/report.pdf.
12. R. Branco and S. Gueron, "Blinded Random Corruption Attacks," *Proc. IEEE Int'l Symp. Hardware Oriented Security and Trust* (HOST 16), 2016, pp. 85–90.
13. "Intel Software Guard Extensions Programming Reference," Intel, 2014; software.intel.com/en-us/isa-extensions/intel-sgx.
14. I. Anati et al., "Innovative Technology for CPU Based Attestation and Sealing," *Proc. 2nd Int'l Workshop Hardware and Architectural Support for Security and Privacy* (HASP 13), 2013; software.intel.com/sites/default/files/article/413939/hasp-2013-innovative-technology-for-attestation-and-sealing.pdf.
15. F. McKeen et al., "Innovative Instructions and Software Model for Isolated Execution," *Proc. 2nd Int'l Workshop Hardware and Architectural Support for Security and Privacy* (HASP 13), 2013, pp. 10:1–10:1.
16. M. Hoekstra et al., "Using Innovative Instructions to Create Trustworthy Software Solutions," *Proc. 2nd Int'l Workshop Hardware and Architectural Support for Security and Privacy* (HASP 13), 2013, pp. 11:1–11:1.
17. S. Johnson et al., *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*, white paper, Apr. 2016; software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.
18. R. Elbaz et al., "Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines," *Trans. Computational Science IV*, Springer-Verlag, 2009, pp. 1–22.
19. M. Henson and S. Taylor, "Memory Encryption: A Survey

of Existing Techniques," *ACM Comput. Surv.*, vol. 46, no. 4, 2014, pp. 53:1–53:26.
20. M.N. Wegman and J. L. Carter, "New Hash Functions and Their Use in Authentication and Set Equality," *J. Computer and System Sciences*, vol. 22, 1981, pp. 265–279.
21. S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," Cryptology ePrint Archive, report 2016/204, 2016; eprint.iacr.org/2016/204.
22. "CINT2006 (Integer Component of SPEC CPU2006)," Standard Performance Evaluation Corporation, 2006; www.spec.org/cpu2006/CINT2006.
23. "Graphene Library OS," Stony Brook Univ.; graphene.cs.stonybrook.edu.

Shay Gueron is an associate professor of mathematics at the University of Haifa, and senior principal engineer at Intel, acting as chief core cryptography architect of the CPU architecture group. His research interests include cryptography, security, and algorithms. Gueron was a member of the team that developed Intel's Software Guard Extensions security technology, was in charge of its cryptographic definition and implementation, and invented the Memory Encryption Engine. He received a DSc in applied mathematics from Technion–Israel Institute of Technology. Contact him at shay@math.haifa.ac.il.