SGX Secure Enclaves in Practice: Security and Crypto Review

JP Aumasson, Luis Merino — Kudelski Security

July 28, 2016

Our Black Hat 2016 talk **SGX Secure Enclaves in Practice: Security and Crypto Review** is the first public report about the security of Intel(R) Software Guard Extensions (SGX) based on actual SGX hardware and on Intel's software development toolchain for Windows and Linux. We describe yet undocumented aspects of SGX and present our new open-source tools released at Black Hat. This report is a non-technical summary of what we've done and discovered, and starts with a high-level introduction to SGX. For more details, refer to our slides and to our projects on GitHub: https://github.com/kudelskisecurity/sgxfun and https://github.com/kudelskisecurity/sgx-reencrypt.

Disclaimer: We are not affiliated with Intel, and information about SGX in this document should not be considered as official or reliable—use it at your own risk. Our research is based on public documentation, on our own analysis of SGX components, and on feedback by Intel.

What's SGX?

Secure Guard Extensions (SGX) is a security technology available since autumn 2015 in Intel CPUs that are based on the Skylake x86 microarchitecture. SGX makes it possible for the first time to use Intel CPUs in order run a program on a remote system—typically a cloud provider—such that

1. The remote system can't alter the program that is executed (integrity)
2. The program can process encrypted data that is decrypted only inside the CPUs, and therefore inaccessible to the remote operating system (confidentiality)

Instead of having to trust the remote operating system and its human administrators, you just have to trust Intel's CPU—which you have to trust anyway, since it's running your code. SGX thus protects an application and its secrets from a full compromise of a remote system as well as from malicious insiders.

SGX achieves this **trusted computing** functionality thanks to **secure enclaves**, or pieces of code and data that can only be executed by a trusted Intel CPU. This required a wholly new security architecture, new complex assembly instructions, a new program model, and a bunch of cryptographic algorithms and protocols. And all these new components must be secure in order SGX' trusted computing functionality to be secure.

A Killer Application

SGX can serve to create executable files that are **impossible to reverse engineer**. Although an enclave's code is not encrypted, it can provide a public key to remote clients, who will send encrypted code to be executed securely. The enclave will then act as virtual machine, taking encrypted bytecode that only it can decrypt.

Everyone's first reaction when hearing this is *"OMG bad guys will use it to create super malware!"*. But it shouldn't be that scary, since:

- Enclave programs are severely limited compared to normal programs: they cannot issue syscalls nor can they perform I/O operations directly.
- Enclave code runs in userland, not in kernel mode. To fully own a system, a privilege escalation from the enclave is needed.
- At the moment, not anyone can build secure enclaves and distribute them to run arbitrarily. Instead, a platform running an enclave's code must verify its attestation with respect to an identified vendor.

Under The Hood

The notion of trusted computing as realized by Intel SGX isn't new. The key concepts actually date back to the 1980s. But it's the first time that a fully fledged trusted computing technology is made available in mainstream CPUs. This is possible thanks to a novel security architecture and its implementation of trusted computing key concepts:

- **Hardware secrets**: Each SGX-enabled CPU includes a unique 128-bit value that even Intel doesn't know. All cryptographic keys derived within a given CPU depend on that key.
- **Remote attestation**: This is the crux of SGX. Remote attestation gives remote users a proof that an enclave is running a given program securely. You can see it as a public key certificate where a trusted entity endorses a program (the enclave) rather than a public key.
- **Sealed storage**: To keep secrets provisioned by a remote user, SGX-enabled CPUs can store these secrets as encrypted blobs outside the CPU,

protecting their confidentiality and integrity.

- **Memory encryption**: Many useful programs will use a larger amount of memory than the CPU's cache and register offer, though other system components may access RAM content using DMA or other mechanisms. SGX thus comes with a dedicated *memory encryption engine*.

SGX Security

The *trusted computing base* (TCB) of SGX consists of the minimal set of components that need to be trusted in order to enjoy SGX' security benefits. The TCB of SGX consists of:

- The **CPU** and all that's inside its packages, including hardware logic, microcode, registers, cache memory.
- The software components used for attestation, in particular the **quoting enclave** (an Intel-built enclave that signs an enclave's measurement within the remote attestation process).

So you've to trust that Intel won't—either deliberately or by accident—hide weaknesses in its CPU, as there's almost no way to verify that a CPU's internal logic is doing what it's supposed to do. Critical software components, however, can partially be analyzed.

Security Guarantees

If SGX behaves as expected, it should protect secure enclaves from any malicious component outside the TCB. In particular, SGX should protect an enclave's code and data from

- The operating system, or hypervisor
- The BIOS, firmware, or drivers of the machine hosting the CPU
- The System Management Mode (SMM) software, a "ring –2"
- The Intel Management Engine (ME) software and hardware, a "ring –3"
- Any remote attacker that would compromise any of the above components

Security Limitations

Security limitations documented by Intel include:

- **Cache-timing attacks**: SGX won't protect a program vulnerable to side-channel attacks that exploit secret-dependent execution time and/or cache access patterns.
- **Physical attacks**: SGX won't protect from attackers that will unpackage a CPU and run attacks such as invasive fault injection attacks, for example.
- **Microcode attacks**: SGX won't protect from attackers that are able to reprogram the functionality of machine code.

Attack Surface

In addition to the intrinsic security limitations of SGX documented by Intel, attackers could compromise the security of an SGX-based application by exploiting:

- Bugs in the enclave software such as classical memory corruption or concurrency issues (SGX will not magically secure insecure software).
- An insecure development environment: if your OS is compromised when developing an enclave then obviously that enclave can't be trusted.
- Unsafe SGX programming by the enclave writer, for example by mishandling in-enclave pointers, or by leaking secret data through function calls outside the enclave.
- Bugs in the Intel trusted libraries (libc or crypto library, for example).
- CPU bugs, in the hardware logic or in the microcode.
- Loopholes in the SGX toolchain supply chain, for example an SDK downloaded over an insecure channel.

Overall, the daunting complexity of SGX raises doubts about its security. Alas (or fortunately?), many potential bugs can only be detected in a black-box fashion, as the code (microcode, hardware) isn't available.

Recovery

To recover from bugs in the microcode implementation of its assembly instructions, SGX microcode can be patched. However, SGX components of pure hardware logic such as the memory encryption engine can't be patched. Likewise, enclaves containing insecure software can't be patched without going through the attestation process again. To recover from a compromise of an enclave's private key (for example), revocation mechanisms are in place to ensure that a given attestation will no longer be valid.

How Strong is SGX Cryptography?

SGX uses cryptography everywhere, be it to sign an enclave, to seal encrypted data off the enclave, or within the intricate remote attestation mechanism. SGX also allows enclaves to use cryptography by providing a cryptography library to enclave authors. But how secure are these algorithms and their implementations?

Random Generation in SGX

Strong randomness is essential to any system doing cryptographic operations. A common mistake in C(++) is to use libc's `rand()` to generate cryptographic material; `rand()` is a non-cryptographic deterministic random bit generator, and will therefore yield predictable cryptographic secrets. We were happy to find that the libc in SGX does not support `rand()`, and instead provides the function

```
sgx_read_rand(unsigned char* rand, size_t length_in_bytes);
```

This function calls the hardware-based pseudorandom generator (PRNG) available in Intel CPUs through the `RDRAND` instruction. You can also call `RDRAND` directly, but it `RDRAND` may fail to return reliable random bytes. It's therefore essential to check the carry flag set to 1 by `RDRAND` when it succeeds. As Intel recommends in its *Enclave Writers Guide*, we observed that `sgx_read_rand()` does attempt to call `RDRAND` up to 10 times before returning an error.

Still, there are caveats with SGX' PRNG:

- `sgx_read_rand()` calls `RDRAND`, not `RDSEED`. The latter instruction is slightly better if you're seeding another PRNG with the random bytes collected.
- `RDRAND` and `RDSEED` are the only non-SGX instructions available to SGX enclaves and that an hypervisor can force to cause a virtual machine exit, through a parameter in the virtual machine control structure. In other words, an hypervisor can prevent an enclave program from using randomness.

Standard Crypto Algorithms

SGX uses many different cryptographic algorithms, including:

- RSA-3072 PKCS#1 v1.5 with SHA-256, to compute enclave signatures
- RSA-2048 PKCS#1 v2.1 encryption (OAEP) with SHA-256, within the attestation process
- ECDSA signatures over the p256 NIST curve, with SHA-256, for launch enclave policy checks

- ECDH over p256, for remote key exchange
- AES-GCM for sealing data
- AES-CMAC for key derivation
- AES-CTR for memory encryption

AES with 128-bit keys and elliptic curve schemes over p256 will get you 128-bit security, however the RSA schemes provide a lower security level (of the order of 112-bit for RSA-3072, and of the order of 96-bit security for RSA-2048).

All of these are trusted, standardized and time-tested algorithms that are unlikely to be broken. Flaws, if any, will likely be in their *implementation*.

For example, software *implementation of AES* are notoriously fragile against cache-timing attacks. In order to counter these, some AES implementations in critical SGX components combine hardware instructions (AES-NI) to perform the series of rounds and a table-based implementation optimized the minimize data leaks. In contrast, the AES implentation made available to Linux developers uses a slower AES implementation without native instructions, yet also with cache-timing mitigations.

Custom Crypto Algorithms

Due to the specific functional and security requirements of SGX, Intel designed custom cryptographic schemes. Perhaps the two most critical are

- **The memory encryption engine** (MME): This protects data stored in RAM with respect to confidentiality, integrity, and replays. The MME uses a dedicated high-speed Carter-Wegman MAC combined with a tweaked AES-CTR. An almost complete description is available at http://eprint.iacr.org/2016/204, along with a rigorous analysis of the scheme's security.
- **The quote creation scheme**: This creates a quote, the analog of a certificate for a secure enclave, where a CPU *signs* a description of an enclave on behalf of Intel. If this scheme is insecure then SGX becomes useless. The underlying cryptographic scheme, however, is not documented.

We therefore tried to find out the cryptographic details behind the quoting scheme. Specifically, we analyzed the `get_quote()` function in the quoting enclave (binary `qe.signed.dll` in the Windows platform software, and its Linux counterpart `libsgx_qe.signed.so`.) In this function, a quote is generated after combining the algorithms RSA-OAEP, AES-GCM, SHA-256, and the actual signature scheme EPID, where AES takes a fresh random key and random nonce and works as in the figure below:

This scheme doesn't look like anything we had ever seen, though it's essentially a hybrid encryption scheme in which RSA encrypts a symmetric key that in turn serves to encrypt and authenticate a signature and other metadata. Unlike for

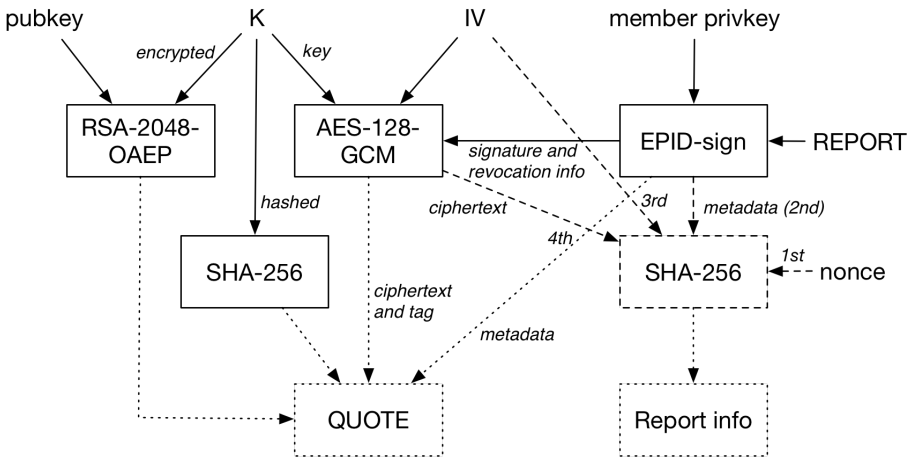

Figure 1: Crypto scheme in the quote generation

the memory encryption engine, the security goals of this scheme aren't specified. We thus can't rigorously define what it means for this scheme to be secure or broken.

Nonetheless, suboptimal properties are a relatively short RSA key, the lack of forward secrecy, and the leak of a hash of the symmetric key that simplifies time-memory tradeoff attacks.

EPID Anonymous Group Signatures

A unique feature of SGX cryptography is its use of an *anonymous group signature* scheme called enhanced privacy ID (EPID). Although EPID is defined in https://eprint.iacr.org/2009/095, there's no detail on the specific version used in SGX let alone on its implementation. We thus tried to gather more information on those.

In SGX, EPID lets an enclave attest that it's running a given software (through its `MRENCLAVE` identity) on a given CPU (model, microcode version, etc.). The use of EPID within SGX is documented in Intel's *EPID Provisioning and Attestation Services* document. EPID isn't a typical signature scheme like ECDSA or RSA-PSS with a signer that signs and a verifier that verifies. Instead, an EPID deployment takes three kinds of entities:

- The **issuer** (Intel) is the only party that knows the *issuing signing key* (ISK), called "Intel Signing Key" in Intel's *EPID Provisioning and Attestation Services* document. The *group public key* (GPK) is published within a group public key certificate signed by the ISK, and verifiable using the ISK's associated public key (which is not the GPK).

- **Members** (CPUs) join the group through an interactive protocol with the issuer, in such a way that the issuer doesn't know the *member private key* SK. Given a enclave's `REPORT` (a measurement that includes the `MRENCLAVE` and `MRSIGNER` identities), a member creates a `QUOTE` that includes an EPID signature of the `REPORT`. The `QUOTE` acts as an attestation that the enclave is running on a trusted SGX platform.
- **Verifiers** are applications running the enclave (locally or remotely), and that will verify that a given `MRENCLAVE` runs on a given trusted computing base.

Some important features of EPID in SGX are:

- **Anonymity**: When a group member issues a signature, no one—even the issuer—can identify the member given the signature. The goal is to prevent the tracking of users through their CPUs.
- **(Un)linkability**: Members have the choice to make signatures either linkable or unlinkable. While it's impossible to tell whether two unlinkable signatures come from the same member, linkable signatures can be grouped by source.

A large part of the EPID implementation is found in the quoting enclave (QE), which EPID-signs a report and creates a `QUOTE` that includes this signature. The QE binary exposes the following EPID member functions:

- `epidMember_create`
- `epidMember_createCompressed`
- `epidMember_delete`
- `epidMember_registerBaseName`
- `epidMember_computePreSignature`
- `epidMember_join`
- `epidMember_isPrivKeyValid`
- `epidMember_signMessagePartial`
- `epidMember_checkSigRLHeader`
- `epidMember_nrProve`
- `epidMember_signMessage`

Specifically, the `get_quote` function in `sgx_get_quote` calls `epidMember_signMessagePartial` to sign a report and `epidMember_nrProve` to generate non-revoked proofs.

As of June 2016, the Linux platform software has been using the following group key certificate, which includes the group key and a signature by Intel's ISK

```
static const uint8_t EPID_GROUP_CERT[] = {
    0x00, 0x00, 0x00, 0x0B, 0xB3, 0x6F, 0xFF, 0x81, 0xE2, 0x1B, 0x17, 0xEB,
    0x3D, 0x75, 0x3D, 0x61, 0x7E, 0x27, 0xB0, 0xCB, 0xD0, 0x6D, 0x8F, 0x9D,
    0x64, 0xCE, 0xE3, 0xCE, 0x43, 0x4C, 0x62, 0xFD, 0xB5, 0x80, 0xE0, 0x99,
    0x3A, 0x07, 0x56, 0x80, 0xE0, 0x88, 0x59, 0xA4, 0xFD, 0xB5, 0xB7, 0x9D,
    0xE9, 0x4D, 0xAE, 0x9C, 0xEE, 0x3D, 0x66, 0x42, 0x82, 0x45, 0x7E, 0x7F,
```

```
    0xD8, 0x69, 0x3E, 0xA1, 0x74, 0xF4, 0x59, 0xEE, 0xD2, 0x74, 0x2E, 0x9F,
    0x63, 0xC2, 0x51, 0x8E, 0xD5, 0xDB, 0xCA, 0x1C, 0x54, 0x74, 0x10, 0x7B,
    0xDC, 0x99, 0xED, 0x42, 0xD5, 0x5B, 0xA7, 0x04, 0x29, 0x66, 0x61, 0x63,
    0xBC, 0xDD, 0x7F, 0xE1, 0x76, 0x5D, 0xC0, 0x6E, 0xE3, 0x14, 0xAC, 0x72,
    0x48, 0x12, 0x0A, 0xA6, 0xE8, 0x5B, 0x08, 0x7B, 0xDA, 0x3F, 0x51, 0x7D,
    0xDE, 0x4C, 0xEA, 0xCB, 0x93, 0xA5, 0x6E, 0xCC, 0xE7, 0x8E, 0x10, 0x84,
    0xBD, 0x19, 0x5A, 0x95, 0xE2, 0x0F, 0xCA, 0x1C, 0x50, 0x71, 0x94, 0x51,
    0x40, 0x1B, 0xA5, 0xB6, 0x78, 0x87, 0x53, 0xF6, 0x6A, 0x95, 0xCA, 0xC6,
    0x8D, 0xCD, 0x36, 0x88, 0x07, 0x28, 0xE8, 0x96, 0xCA, 0x78, 0x11, 0x5B,
    0xB8, 0x6A, 0xE7, 0xE5, 0xA6, 0x65, 0x7A, 0x68, 0x15, 0xD7, 0x75, 0xF8,
    0x24, 0x14, 0xCF, 0xD1, 0x0F, 0x6C, 0x56, 0xF5, 0x22, 0xD9, 0xFD, 0xE0,
    0xE2, 0xF4, 0xB3, 0xA1, 0x90, 0x21, 0xA7, 0xE0, 0xE8, 0xB3, 0xC7, 0x25,
    0xBC, 0x07, 0x72, 0x30, 0x5D, 0xEE, 0xF5, 0x6A, 0x89, 0x88, 0x46, 0xDD,
    0x89, 0xC2, 0x39, 0x9C, 0x0A, 0x3B, 0x58, 0x96, 0x57, 0xE4, 0xF3, 0x3C,
    0x79, 0x51, 0x69, 0x36, 0x1B, 0xB6, 0xF7, 0x05, 0x5D, 0x0A, 0x88, 0xDB,
    0x1F, 0x3D, 0xEA, 0xA2, 0xBA, 0x6B, 0xF0, 0xDA, 0x8E, 0x25, 0xC6, 0xAD,
    0x83, 0x7D, 0x3E, 0x31, 0xEE, 0x11, 0x40, 0xA9
};
```

The ISK's public key, used to verify a signature, is:

```
/* This is the x component of production public key
   used for EC-DSA verify for EPID Signing key. */
const uint8_t g_sgx_isk_pubkey_x[] = {
    0x26, 0x9c, 0x10, 0x82, 0xe3, 0x5a, 0x78, 0x26,
    0xee, 0x2e, 0xcc, 0x0d, 0x29, 0x50, 0xc9, 0xa4,
    0x7a, 0x21, 0xdb, 0xcf, 0xa7, 0x6a, 0x95, 0x92,
    0xeb, 0x2f, 0xb9, 0x24, 0x89, 0x88, 0xbd, 0xce
};

/* This is the y component of production public key
   used for EC-DSA verify. Same as upper. */
const uint8_t g_sgx_isk_pubkey_y[] = {
    0xb8, 0xe0, 0xf2, 0x41, 0xc3, 0xe5, 0x35, 0x52,
    0xbc, 0xef, 0x9c, 0x04, 0x02, 0x06, 0x48, 0xa5,
    0x76, 0x10, 0x1b, 0xa4, 0x28, 0xe4, 0x8e, 0xa9,
    0xcf, 0xba, 0x41, 0x75, 0xdf, 0x06, 0x50, 0x62
};
```

To sign a `QUOTE`, the quoting enclave gets the ISK private key via the provisioning enclave.

Internally, EPID relies on bilinear pairings on elliptic curves to achieve the anonymous group signature functionality. In SGX, it does so using *optimal Ate pairings*, a kind of pairing that minimizes the number of iterations of the Miller loop, the main component of a pairing computation. The curve used is allegedly a Barreto-Naehrig curve, along the lines of the fast pairing implementation described in the paper at http://eprint.iacr.org/2010/354.

Software Released

At Black Hat we're releasing two software projects

- **Proxy reencryption enclave**: This proof of concept showcases an SGX-based application that receives data encrypted under some key K1 and securely transforms it to the same data encrypted with some key K2, where the keys are securely stored by the SGX-enabled CPU and subject to predefined permissions. Symmetric key-based systems can use this to avoid complex key management processes and technologies, and to avoid having to trust operating systems and human operators. The source code of this project is available at https://github.com/kudelskisecurity/sgx-reencrypt.
- **Metadata extraction tools**: We wrote Python scripts to parse documented structures of SGX binaries, such as enclave files or attestation quotes. These are available at https://github.com/kudelskisecurity/sgxfun.

Acknowledgments

We are grateful for their help and feedback to Victor Costan (MIT), Shay Gueron (Intel), Simon Johnson (Intel), Samuel Neves (University of Coimbra), Joanna Rutkowska (Invisible Things Lab), Arrigo Triulzi, Dan Zimmerman (Intel).

Biographies

Jean-Philippe (JP) Aumasson is Principal Cryptographer at Kudelski Security, in Switzerland. He designed the popular cryptographic functions BLAKE2 and SipHash, and the new authenticated cipher NORX. He gave talks at Black Hat, DEFCON, RSA, CCC, SyScan, Troopers. He initiated the Crypto Coding Standard and the Password Hashing Competition projects, and co-wrote the 2015 book "The Hash Function BLAKE". JP tweets as @veorq.

Luis Merino is Senior Security Engineer at Kudelski Security, Switzerland working on research projects. In the past, he has been involved in engineering and research projects at Riscure, the Andalusian Astrophysics Institute, and the University of Granada, amongst others. He graduated in computer engineering at University of Granada and is Offensive Security certified. Luis tweets as @iamcorso.