General overview of Intel SGX

Julian Schütz

ABSTRACT

Intel Secure Guard Extensions is a hardware feature available to Intel processors that provides a trusted execution environment fit for remote computation. It enables the creation of secure enclaves, handled by the trusted processor, that are isolated from all other software including the host's system software. They also provide means for users to remotely attest the legitimacy of the enclaves. The mechanisms that were developed to achieve this will be presented in this overview.

KEYWORDS

trusted computing, intel sgx, secure guard extensions

1 INTRODUCTION

Today, software is commonly deployed on remote computers not owned by a trusted party, like in Infrastructure as a service environments such as Amazon Web Services. Such an untrusted party can attempt to steal sensitive data from platforms they have access to. These developments come with considerable security risks since the remote computer is owned by an untrusted party. The use of such services has become so widespread that the need for a way to trust the remotely running code and the integrity of remotely stored secrets has brought new technologies to life.

Intel Secure Guard Extensions (SGX) is a set of hardware extensions for x86 systems Intel has developed with the aim to enhance the security of sensitive data and applications in environments with untrusted system software. Even on compromised systems, the Intel processor as a trusted component enables the concept of an "enclave", a memory region isolated from all other software that cannot be accessed without explicitly being allowed by the trusted hardware to do so [10]. SGX's trusted computing base is limited to only the processor's microcode and the code running in a few special enclaves [8]. SGX enclaves can be remotely attested to be working correctly before receiving their sensitive data and code through the use of cryptographic signatures with Intel-specific keys [14]. Multi-core processors are fully supported in SGX enclaves [8] to keep performance high for complicated computations.

With SGX being available in consumer processors from Intel's 6th generation in 2015 up until the 10th generation and on Intel server processors since 2015, SGX has established itself as a mainstay in security. It provides a new standard for secure applications that can and is being expanded upon to bring its security to more applications. Intel's focus on making it simple to integrate SGX into an existing application has resulted in widespread adoption by security sensitive software.

2 BACKGROUND

In this section, the background concepts that SGX builds upon are introduced.

2.1 Software Privilege Levels

In x86 architecture, software typically runs in four privilege levels, from ring 0, the most privileged, to ring 3, the least privileged. Any privilege level is strictly more powerful than the levels below it, allowing software to read or write data at those levels [9]. When user applications, which run at ring 3, require services of more privileged software, like accessing privileged memory, they need to call well defined functions that switch the privilege level internally. Software can not be allowed to freely switch privilege levels to uphold security standards expected by modern applications. [8]

SYSCALL is the primary instruction software on modern processors can call to use ring 0 code. A *SYSCALL* jumps into ring 0 code at a location defined by special model specific registers, referred to as system call handler, which cannot be accessed outside of ring 0 code. *SYSRET* then switches the privilege level back and continues the execution of ring 3 code [10].

2.2 Threat model

The adversaries SGX aims to protect against include an unprivileged software adversary, a system software adversary and a startup code adversary [4].

The unprivileged adversary or "ring-3" attacker is limited by the privileges granted to him by system software. He can only execute ring-3 instructions and perform read or write operations to memory mapped with read or write permissions given by system software [4]. The system software attacker is in control of the operating system and can read or write to all available memory. This malicious actor is capable of scheduling code execution as well as executing a potentially malicious SGX enclave [4]. The startup code adversary has compromised the BIOS and as such, has full control over the platform during startup and is able to modify registers that are exclusive to the system management mode [4]. A potential attacker in SGX's adversary model has full physical access to the platform the enclave is running on and as such, is able to see and store all changes to the DRAM.

Importantly, this list of threats does not include a side-channel adversary, who is able to evaluate various statistics about the processor to draw conclusions about the software being executed. Some of these statistics include the power draw of the processor, cache misses, branches and DRAM accesses [4]. Availability threats like denial of service attacks are not covered by SGX, as an attacker with physical access to the platform can shut it down at any time.

3 ARCHITECTURE

A Intel SGX Enclave resides in an environment isolated from untrusted software and hardware. Its mechanisms protect the integrity of the code and the data inside the enclave while keeping the process of implementing SGX into applications for developers as simple as possible. This chapter summarizes the mechanisms in place that allow SGX to fulfill those goals.

3.1 Memory Layout

Intel SGX enabled processors store all data relevant to enclaves in the Processor Reserved Memory (PRM). This memory is a portion of DRAM not accessible to other software including system software. Even if the DRAM were to be removed and read out on another platform, the contents of the PRM would not be leaked as they are encrypted with a key stored directly in the processor. This key changes every power cycle to reduce consequences in case it is compromised [10]. The processors memory controller protects the data from being accessed by peripherals by denying direct memory access attempts into the PRM [10].

Utilizing the regular page tables that applications use would come with major security flaws, which is why SGX uses the Enclave Page Cache (EPC). The EPC resides within the PRM, restricting access even for system software. Inside the EPC, memory is split in pages of 4 KB in size, each of which can only be assigned to one enclave at a time [10]. A EPC translates a virtual address to a physical memory address similarly to regular page tables.

The EPC can only be accessed from inside an enclave, which is why the same SGX instruction handles page allocation and page initialization, usually consisting of copying from a regular page [8]. Because the untrusted system software allocates the EPC pages to enclaves, the trusted hardware needs to check that the EPC entries are not allocated incorrectly. For this reason, the processor keeps a record of allocations for each EPC in the Enclave Page Cache Map (EPCM). If access to any EPC is requested from unauthorized software, the processor will issue a fault and deny access [8].

An entry in the EPCM consists of a valid bit that is set to 0 for unallocated pages and to 1 for all others and 8 bits specifying the page type which are set upon allocating and initializing the enclave [10]. Depending on the type, the page stores an enclave's code and data or a special SGX data structures like the SGX Enclave Control Structure. The identity of the enclave owning the entry in the EPCM is also stored inside it, making it possible to prevent an enclave from accessing other enclaves pages and their data [8].

3.2 SGX Enclave Control Structure

The SGX Enclave Control Structure (SECS) is an assortment of metadata about one specific enclave. An enclave's SECS is the primary form of identification used by SGX. An enclave's SECS is stored in a dedicated EPC page with a dedicated page type associated with it. The enclave identifying information in an EPCM entry is a pointer to the enclave's SECS [8], making it and its contents one of the most important aspects of SGX's architecture.

A SECS contains the identities `MRENCLAVE` and `MRSIGNER`, as well as `SSAFRAMESIZE`. `MRENCLAVE` is a measurement of the enclaves startup environment used to identify contents of an enclave and `MRSIGNER` is a measurement of the "Sealing Authority". `SSAFRAMESIZE` specifies how many pages are used to store the context in an enclave's State Save Area (SSA). It also contains a value of `BASEADDR`, `SIZE`, the `ATTRIBUTES` associated with the enclave, as well as the enclave's product ID `ISVPRODID` and its security version number `ISVSVN` [8].

The security version number can distinguish two enclaves even if the product ID, `MRENCLAVE` and `MRSIGNER` are the same value. This concept allows SGX developers to perform security patches for compromised enclaves by releasing a version of their product with a higher security version number. SGX will deny any attempt to migrate data from an enclave to another one with the same value of `SVNPRODID` and `MRENCLAVE`, but a lower version value in `ISVSVN`. This means that after a user has upgraded and the data has been migrated to the new version, attacks based on the old vulnerable enclave will cease to work [8].

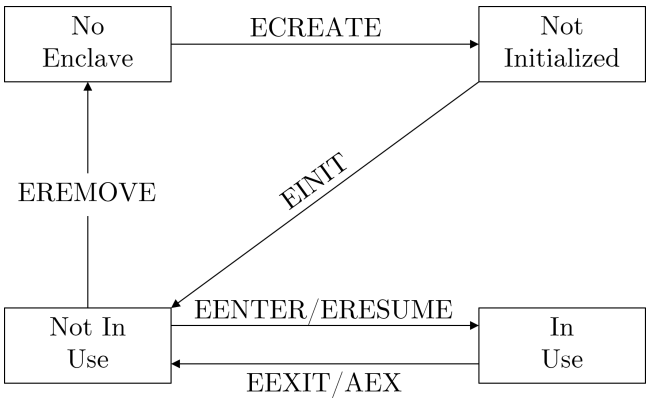

Figure 1: Simplified Lifecycle Of A SGX Enclave

3.3 SGX Thread Control Structure

SGX provides full support for multi-threading, allowing multiple threads to work on the same code of the same enclave at the same time [8]. To achieve this, SGX uses the Thread Control Structure (TCS), one for each logical processor working on the code of an enclave [10]. Similar to the SECS, a TCS is stored in a EPC with a specific page type.

The contents of a page of this type can only be directly read by SGX instructions or special debugging instructions [10], which will not be enabled during normal use. The TCS's contents contain addresses for context switching in or out of the enclave. The entries specify the address to be loaded into Thread Local Storage and the value loaded in the instruction pointer, specifying exactly where and with what data a thread starts executing code inside an enclave [8].

3.4 Enclave Lifecycle

As seen in Figure 1, a SGX Enclave starts to exist when *ECREATE* is called by an application, allocating an unused page in the EPC for the SECS of the new enclave. *ECREATE* also ensures that the values used for initializing the SECS are valid, allowing all SGX instructions that are called later to assume the values in the enclave's SECS are correct [8]. At this time, the enclave can load necessary code, data and TCS pages.

When the enclave is fully prepared, *EINIT* sets the enclaves SECS to be initialized, prohibiting further loading of data and allowing it to be executed [8].

An application can call *EREMOVE* to destroy an enclave by freeing the EPC page storing the enclave's SECS. It will however only do so if the SECS page is not referenced by any other EPC page, making this the final step in destroying an enclave. Before this, all

other EPC pages relevant to the enclave are set to no longer be a valid page, which also marks them ready for a new allocation. *EREMOVE* checks that no processor is currently executing code in the EPC pages before freeing them, blocking any attempt to destroy an enclave while it is in use [10].

The details of the process of entering and exiting the enclave is vital to SGX's security guarantees, which are discussed now.

3.4.1 Entering An Enclave

EENTER sets the currently active thread into *enclave mode*, allowing it to access the standard EPC pages owned by the enclave the thread is working on. *EENTER* does not change the privilege level, meaning that the enclave code is executed at ring 3, the same privilege level of the application that uses the enclave [11]. This feature prevents potentially malicious enclaves from damaging the host platform more than a regular user application, since they share the same privilege level [8].

While in enclave mode, hardware breakpoints and other debugging features of the processor which would allow leakage of the enclave's data are disabled [8]. *EENTER* requires that the TCS used as its argument is not already in use and that the TCS still has one or more State Save Areas marked as available. This SSA will be used in the case of an Asynchronous Enclave Exit (AEX), which is detailed in subsection 3.4.2. *EENTER* loads data from the segment register specified in the TCS and ensures that the enabled architectural features are exactly as detailed in the enclave's `ATTRIBUTES` by setting the associated register [8]. Additionally, similarly to a *SYSCALL*, *EENTER* backs up the old values of all edited registers to allow restoration of the state after the enclave is exited again.

ERESUME works very similar to *EENTER*, with the important difference being that *EENTER* requires a SSA to be completely empty while *ERESUME* works with a SSA that has been changed by an AEX, failing if the State Save Area is empty [10].

3.4.2 Exiting An Enclave

The processor stores the state of a thread when it leaves the enclave in special secure memory called State Save Area. This allows hardware exceptions to function while a thread is executing code inside an enclave while still maintaining adequate security and prohibiting system software from reading the enclave's data.

SSAs are pointed to in the TCS pages, each SSA using as many pages as the value of `SSAFRAMESIZE` within the enclave's SECS dictates [8]. The TCS specifies the virtual location of the first SSA for that TCS as well as how many SSAs the TCS supports [10].

EEXIT only functions while the thread calling it is in *enclave mode* and sets it to not be in *enclave mode* anymore while loading the values stored by *EENTER* back into their associated registers. It does not clear the registers used by the enclave's code meaning that enclave developers need to properly remove any traces of data that they used [8].

In the scenario that the thread needs to exit the enclave's code without calling *EEXIT* itself, like an interrupt, it is important to not call the regular system software exception handler immediately like most applications would do since that handler is not trusted. For this reason, the processor executes a special AEX [10]. The AEX saves the complete state of the thread executing the enclave's code in its SSA, marking it as used, and restores the information backed up by *EENTER*. To prevent the system software from accessing secrets, all registers relevant to the enclave are cleared [10]. After the system software is done handling the exception, the thread will jump into an asynchronous exit handler inside the process that also hosts the enclave. It is expected for this handler to then continue execution of the enclave by calling *ERESUME* [8].

3.5 Enclave and Sealing Identity

To provide identities for attestation and sealing, every SGX enclave has two measurement registers, which only the trusted computing base has write access to. `MRSIGNER` provides the authority's identity and `MRENCLAVE` provides the enclave's identity.

`MRENCLAVE` is the measurement of the environment an enclave was initialized in, being the contents of the pages, the position of the pages in the enclave and all page-associated security flags [12]. The information resides in the register in the form of a SHA-256 hash. The value of `MRENCLAVE` is immutable after calling `EINIT`.

`MRSIGNER` is used to store a signed key of the "Sealing Authority". The "Sealing Identity" that is used to provide data integrity contains a version number, a product ID and the Sealing Authority, the entity that signs the enclave beforehand [3]. The Sealing Authority, usually the enclave builder, provides the hardware with a signed enclave certificate including the public key of the Sealing Authority as well as the expected value of `MRENCLAVE` [4]. The trusted hardware can then confirm that the value of `MRENCLAVE` is as expected by comparing the two. Only if this check passes, the SECS of the enclave will be filled with the values of the certificate and the public key provided by the Sealing Authority will be hashed and stored in `MRSIGNER` [8].

3.6 Sealing

When a SGX enclave exits, it is destroyed entirely as discussed in subsection 3.4.2. If a SGX developer wants to retain data even after an enclave is removed, special sealing instructions are required to ensure that the data is kept safe. For this purpose, *EGETKEY* provides access to a "Sealing Key" which can encrypt enclave data. Later, the sealed data can be decrypted by another call to *EGETKEY*. In combination with additional measures offered by Intel SGX platform services [7], such as monotonic counters, this process can be protected against replay attacks [3].

EGETKEY requires a policy which decides what enclaves will later be able to decrypt the sealed data. The two policies decide between sealing to the Enclave Identity or the Sealing Identity.

Sealing to the Enclave Identity means the key used to encrypt the data is a result of the calling enclave's value of `MRENCLAVE`. This policy creates a different sealing key for every enclave, as any change of `MRENCLAVE` will lead to a different sealing key. As such, data sealed to the Enclave Identity can only be unsealed by this exact enclave, even after the enclave is destroyed and created again. This policy proves useful if there will be scenarios in which the SGX developers does not want data to be reused after an update, like authentication tokens [3].

When using the policy to seal to the Sealing Identity, the key is instead influenced by the enclave's `MRSIGNER` and `ISVSVN` value. Since `MRSIGNER` represents the Sealing Identity, multiple enclaves signed by the same authority are allowed to decrypt data that was encrypted by others [3].

ISVSVN contains the security version number, which should differ from the product version number. Careless increments of the security version number could reach the maximum of 65,536 combinations the 2 bytes allocated for ISVSVN can represent [8]. This value should only be updated if a security vulnerability has been addressed in the update.

The Seal Key is related to a specific security version number which the enclave can choose upon calling *EGETKEY*. Enclaves with a ISVSVN value lower than what was used to seal the data will not be allowed to decrypt it. On the other hand, an enclave with any security version number higher than what the data was originally sealed with is allowed to decrypt it. This policy allows updated versions to seamlessly continue unsealing data which will become unavailable to outdated versions once it is resealed with a higher security version number [3].

To prevent the access of an entire platforms secrets when it changes ownership, SGX includes a register called OWNEREPOCH. The value of this register is always part of the key derivation process offered by *EGETKEY* [8]. During normal use, it can be ignored. Since changing the value of OWNEREPOCH leads to different key results in *EGETKEY* without changing any functionality of a enclave [3], this register presents the mechanism to make all secrets on a platform unavailable without the need to remove them. This is especially helpful when ownership of a platform changes only temporarily like in the case of hardware maintenance. Simply changing the value of OWNEREPOCH back to its old value will immediatly make all secrets stored on the platform available again [3].

3.7 Attestation

SGX offers attestation that allows a remote computation service user as well as other enclaves on the same platform to confirm that the software has been correctly instantiated on the remote computer. An attestation contains the identity of the software that is being attested, details of all non-measurable states, like the mode of the running software, as well as data associated with the software and a cryptographic signature representing the trusted computing base [18].

3.7.1 Local attestation

SGX enclaves are able to communicate with each other e.g. in order for developers to be able to construct a higher level protocol for which one application using SGX is not sufficient. The *EREPORT* instruction is used for authentication purposes between two SGX enclaves. When called, *EREPORT* creates a REPORT structure which consists of a message by the enclave, its ATTRIBUTES set during *ECREATE*, the values of MRENCLAVE and MRSIGNER and a message authentication code (MAC) [8]. The MAC is a cryptographic binding over the REPORT calculated with a symmetric "Report Key". Importantly, this key is shared between the enclave receiving the REPORT and the processor via *EREPORT* [3]. Thus, a REPORT is specific to exactly two SGX enclaves and will not be valid if it reached a different, third enclave.

After communication through an untrusted channel has been established, Enclave B can send its MRENCLAVE, as shown in Figure 2. Enclave A passes this value to *EREPORT*, which will then be able to sign the REPORT's MAC with Enclave B's report key. Enclave A then sends the REPORT over the untrusted path [3]. Enclave B

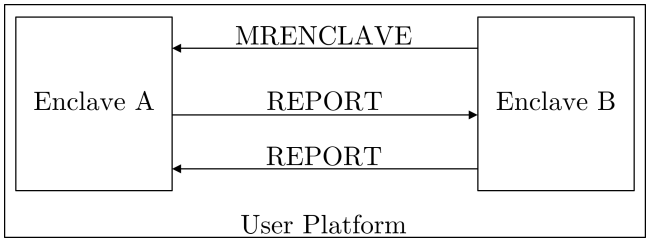

Figure 2: Local Platform Attestation Example

can now call *EGETKEY* and recalculate the MAC that came with the REPORT to confirm that it has not been tampered with and that Enclave A is running on the same platform as Enclave B. At this step, Enclave B can read the values of MRSIGNER to confirm Enclave A's sealer's identity and MRENCLAVE to learn which software is running in Enclave A, as well as pass it on to *EREPORT* [8]. This REPORT will be sent to Enclave A, which can then also confirm that both enclaves are running on the same platform, establishing trust between the two.

3.7.2 Remote attestation

Remote attestation allows a party to be confident that the expected software is running inside a SGX enclave. It can be performed as soon as the enclave is instantiated. At this time, the enclave and the service provider will set up an authentication token, which the enclave will store in its encrypted state. The key used for this encryption, the "Seal Key", is unique to the enclave on the specific platform and the trusted computing base. The sealed authentication token can be used to confirm that the trusted computing base did not change after an application restart [14].

SGX provides a special enclave, called Quoting Enclave, which handles remote attestation via asymmetric cryptography. The Quoting Enclave gathers local attestation REPORTs of SGX enclaves and creates a signature using Intel Enhanced Privacy ID (EPID), Intel's extension to the Direct Anonymous Attestation algorithm [18]. EPID signatures are completely anonymous, even to Intel. As an alternative to EPID, Intel offers Data Center Attestation Primitives, which are further discussed in subsection 3.7.4.

Figure 3 showcases the procedure of a standard remote attestation using SGX with the Quoting Enclave. After a remote challenger requests attestation about the SGX enclave from the user application hosting the enclave (1), the hosting process passes the request on towards the enclave together with the MRENCLAVE value of the Quoting Enclave (2) [18]. The enclave then creates a local attestation with the Quoting Enclave acting as the target enclave and sends it to the user application (3), which passes it to the Quoting Enclave (4). The Quoting Enclave then confirms the correctness of the REPORT by recalculating the MAC as discussed in subsection 3.7.1 before signing it using EPID [14] and sending it back to the user application (5). The remote challenger then receives the remote attestation (6) and can verify it through communication with the Intel Verification Service (7, 8) [8]. Now, the remote challenger has successfully confirmed that the enclave has correctly been instantiated.

The EPID private key used by the Quoting Enclave to sign a remote attestation is a product of the Root Provisioning Key of

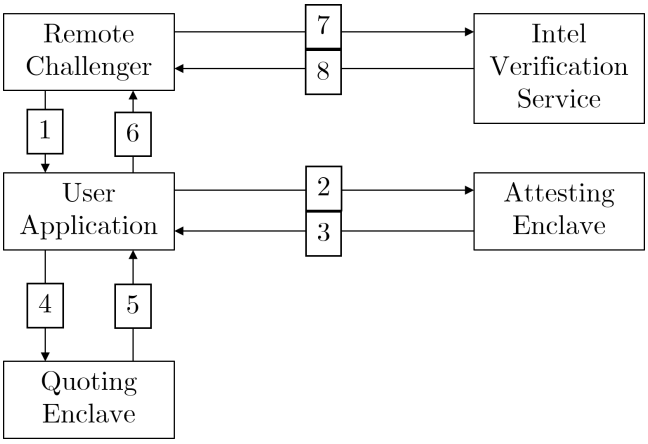

Figure 3: Remote Attestation Example adapted from [18]

the processor and a Intel provisioned secret. The generation of the private key takes place inside a special Provisioning Enclave [14]. The Provisioning Enclave receives a Provisioning key from the processor and then uses this key to aquire an attestation key used for remote attestation from Intel. The attestation key is then encrypted using a special key accessed with *EGETKEY* called Provisioning Seal Key. When the Quoting Enclave creates a remote attestation, it decrypts the attestation key using the Provisioning Seal key via *EGETKEY*, replaces the MAC with a signature created with the attestation key and re-encrypts it [8].

EPID group signatures have the unique property of being completely anonymous, even to the group issuer itself. This means it is impossible for any entity to determine which keys were used in the creation process of the signature [14].

Two distinct signature modes are supported by EPID: Random Base Mode and Name Base Mode. In Random Base Mode, the signing entity chooses a random base for every signature, making it impossible to verify later that all its signatures were done by the same signer with the same key. In Name Base Mode, the signer picks a base to use for all its signatures, which means it is possible to link all its signatures together [14].

The information gained by generally using Name Base Mode can be used in order to minimize the impact of compromised keys. To showcase this, consider a party using SGX remote attestation to authenticate login tokens. If an EPID key is compromised, the adversary can use it to sign any number of malicious logins. In Random Base Mode, it is not possible to distinguish the malicious logins from valid ones. In Name Base Mode however, the party using SGX can check if the logins were signed using the same key. This makes it possible to categorize a suspicious amount of logins signed by the same key as malicious and revoke all logins that used the compromised key. In addition, the party can find out which customer originally used the now compromised key and notify them that their platform has most likely been infected with malware. For this reason, it is recommended to generally use Name Base Mode [14].

3.7.3 *Revocation*

Intel uses the fact that they have a list of public keys for revocation purposes. EPID provides similar revocation methods to traditional signing algorithms like RSA with some distinct differences [14]. In Name Base Mode, it is possible to revoke all signatures performed by a signing entity. Intel also provides a form of signature based revocation with EPID regardless of the Mode used.

If Intel's revocation authority receives a signature that is deemed suspicious enough to revoke, it is added on the signature revocation list. This does not simply revoke the one signature, but has the effect of revoking all signatures created by the same key as the compromised signature [14]. Any attestation calculates a "non-revoked proof", a little documented process that attests to the verifying party that the attesting platform did not generate a specific signature on the signature revocation list. This process is repeated for every entry on this list [14].

3.7.4 *Data Center Attestation Primitives*

When deploying SGX at a large scale, like in Data Centers, it is often impractical for every platform to regularly communicate with Intel directly. For cases such as offline environments, entities that outsource trust decisions and others, Intel Data Center Attestation Primitives provides infrastructure that allows third parties to locally certify Quoting enclaves while still maintaining the certificate chain that leads to Intel [18].

To achieve this, Intel offers a special enclave called the Provisioning Certification Enclave (PCE). This enclave works with two values available via *EGETKEY*. The first is the Provisioning Certification Key (PCK), which is unique to the current trusted computing base. The second is the Platform Provisioning ID, which is only unique to the platform [18].

As there will not be continuous contact with Intel, the Quoting Enclave used in this scenario does not use Intel's EPID, but instead uses Elliptic Curve DSA encryption. This form of encryption does not need a confirmation from a back-end server to be certified as being valid. The local Quoting enclave can request certification by providing the PCE with the attestation public key. The PCE then signs a certificate that identifies the Quoting Enclave and the attestation public key with the PCK. Intel publishes certificate revocation lists in all Intel platforms, completing the certificate chain [18].

4 DEVELOPMENT EXPERIENCE

Developing an application that uses SGX requires Intel's SGX software development kit (SDK), which is available for download for Windows and Linux operating systems [1] [2]. SGX enabled hardware includes most Intel Core processors released between 2015 and 2020 as well as most server processors released since 2015. The Linux kernel supports SGX instructions in release versions 5.11 and newer [2].

Due to the storage size limitation in the PRM, it is recommended to keep the space used by enclaves as low as possible [10]. This means additional work for developers who have to partition an application to keep the security sensitive part running inside a SGX enclave small. This step needs to be done manually, preventing developers from porting existing applications into SGX in bulk.

Releasing a product that includes the use of SGX enclaves also requires whitelisting from Intel directly. This includes the need to sign a Commercial Use License Agreement with Intel [13], only

then will Intel allow a new signer to be added to the SGX Whitelist. This process allows Intel to effectively control what entities are allowed to use the SGX technology, which has been a reason for criticism [8].

5 RELATED WORK

Previously, the problem of trusted code execution has been tackled by other works like Flicker [16]. Flicker boasts an exceptionally small trusted computing base with only as few as 250 lines of code while also providing remote attestation features and protection against a malicious BIOS or operating system. It differs from SGX as it stops all code execution while a sensitive piece of code is executed [16]. This creates a large performance overhead especially on multi-core processors where SGX excels.

SGXoMeter [15] has presented a standardised tool for benchmarking performance of SGX. For large data sizes, the performance overhead created by SGX generally becomes negligible. For smaller sizes, different versions of the SGX SDK perform noticeably different, trending towards lower performance. This might be due to the additional mitigation features to protect against certain attacks that were patched in the newer versions of the SGX SDK [15]. Version 2.12 of the SDK decreased the performance of the SHA-256 algorithm by a factor of about 10 [15].

As SGX provides a baseline of enclave systems, many projects have improved by implementing the use of SGX enclaves with relatively small performance losses and great security gains. One of these projects is SecureKeeper [6], which was developed as a version of the coordination service Apache ZooKeeper with SGX implementation. ZooKeeper is a service that provides its users tolerance of crashes by offering several replicas that are all connected [6]. SecureKeeper takes the design of ZooKeeper and implements multiple enclaves which guarantee that the user data will always be encrypted while keeping all regular functionality and a similar performance overhead to using secure channels instead [6].

SGX has not proven to be infallible, as numerous exploits are described in [20]. It is possible for an attacker to infer sensitive data, e.g. by tracking accesses to pages. As Xu et al. [21] have shown, page accesses can be monitored by unmapping pages, causing a page fault when a enclave attempts to access them. Developers are also still responsible to protect their application against memory corruption attacks like buffer overflows. Especially in an environment constructed to handle sensitive data, bugs such as these are prime targets for attackers [20].

SGAxe [19] shows how transient execution attacks can abuse false branch predictions and speculative execution done by processors leading to the leakage of sensitive data, including the supposedly secure data inside a SGX enclave. PlunderVolt [17] details how fault injection attacks by controlling the processor's voltage can target *EREPORT* and *EGETKEY* in order to extract cryptographic keys.

In the case of an architectural bug like in the case of the Æpic leak [5] in which an attacker with administrator permissions can leak data from the processors cache, SGX is as vulnerable as other software, especially since it is very common for an attacker to have physical access to the platform in SGX's adversary model.

6 CONCLUSION

Intel SGX provides a system that allows data to be securely computed with and stored on remote platforms within an enclave in the processor. As the processor is trusted hardware, even compromised operating systems cannot simply get access to the secrets kept inside a SGX enclave. This combined with the relative ease of implementing SGX into existing applications makes it an appealing solution to parties that commonly use remote computation services.

While SGX does not provide security guarantees in every way, most significantly to side-channel attacks, it is a major improvement in security functions for many parties with often reasonable performance trade-offs.

SGX has shown to lay a promising foundation for the future of security in applications with many projects being built upon SGX as the groundwork.

REFERENCES

[1] 2017. *Getting Started with Intel Software Guard Extensions SDK for Microsoft Windows OS.* Retrieved 07 January, 2023 from https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-sgx-sdk-for-windows.html

[2] 2023. *Intel Software Guard Extensions SDK for Linux OS.* Retrieved 07 January, 2023 from https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/linux-overview.html

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Aug. 2013. Innovative technology for CPU based attestation and sealing.

[4] I. Anati, F. McKeen, S. Gueron, H. Huang, S. Johnson, R. Leslie-Hurd, H. Patil, C. V. Rozas, and H. Shafi. Tutorial Slides presented at ICSA 2015. Intel Software Guard Extensions (Intel SGX), 2015.

[5] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*.

[6] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Proceedings of the 17th International Middleware Conference* (Trento, Italy) *(Middleware '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/2988336.2988350

[7] Shanwei Cen and Bo Zhang. 2017. Trusted Time and Monotonic Counters with Intel® Software Guard Extensions Platform Services. https://cdrdv2-public.intel.com/671564/intel-sgx-platform-services.pdf.

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf.

[9] Johan De Gelas. 2008. *Hardware Virtualization: the Nuts and Bolts.* Retrieved 07 January, 2023 from https://www.anandtech.com/show/2480

[10] Intel. 2015. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[11] Intel. 2016. Overview of an Intel Software Guard Extensions Enclave Life Cycle. https://www.intel.com/content/dam/develop/external/us/en/documents/intelsgxenclavelifecycle.pdf.

[12] Intel. 2016. Overview of Intel® Software Guard Extension Enclaves. https://www.intel.com/content/dam/develop/external/us/en/documents/overview-of-intel-sgx-enclave-637284.pdf.

[13] Intel. 2018. Overview on Signing and Whitelisting for Intel® Software Guard Extension (Intel® SGX) Enclaves. https://www.intel.com/content/dam/develop/external/us/en/documents/overview-signing-whitelisting-intel-sgx-enclaves-737361.pdf.

[14] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. March 2016. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. https://www.intel.com/content/www/us/en/content-details/671370/intel-sgx-intel-epid-provisioning-and-attestation-services.html.

[15] Mohammad Mahhouk, Nico Weichbrodt, and Rüdiger Kapitza. 2021. SGXoMeter: Open and Modular Benchmarking for Intel SGX. In *Proceedings of the 14th European Workshop on Systems Security* (Online, United Kingdom) *(EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 55–61. https://doi.org/10.1145/3447852.3458722

[16] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for Tcb Minimization. In

Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Glasgow, Scotland UK) *(Eurosys '08)*. Association for Computing Machinery, New York, NY, USA, 315–328. https://doi.org/10.1145/1352592.1352625

[17] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1466–1482. https://doi.org/10.1109/SP40000.2020.00057

[18] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Tmijewski. 2016. Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitive. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sgx-support-for-third-party-attestation-801017.pdf.

[19] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX Fails in Practice. https://sgaxeattack.com/.

[20] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. 2022. SoK: SGX.Fail: How Stuff Get eXposed. https://sgx.fail.

[21] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656. https://doi.org/10.1109/SP.2015.45

Received 09 January 2023