Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Hardening cryptographic operations through the use of secure enclaves

André Brandão[*], *João S. Resende*, *Rolando Martins*

CRACS/INESC-TEC, C3P & Faculty of Sciences, University of Porto, Portugal

ARTICLE INFO

Article history:
Received 12 November 2020
Revised 19 April 2021
Accepted 14 May 2021
Available online 24 May 2021

Keywords:
Security
Key management
Trust
Trusted execution environment
Intel SGX

ABSTRACT

With the rising popularity of the cloud, companies lose control of both the hardware and the operating system responsible for hosting their software and data. This means that companies are at risk of losing confidential data when these are utilized in components controlled by a third-party cloud vendor. Secure enclaves can help solve this problem by creating a secure environment where code can be executed securely, guaranteeing that no unwanted parties read or modify the data inside this secure environment.

While the use of secure enclaves has been focused on small footprints software, such as the implementation of trusted computing base for distributed protocols, we analyze the strengths and shortcoming of current tools in an effort to further expand the applicability of their use. Given the importance of web servers and their inherent greater exposure to attacks, we explore the hardening of Apache web server through the use of secure enclaves. This was accomplished by making the necessary modifications to further protect its private key from both the operating system and hypervisor. We also provide a performance assessment to quantify the overhead associated with the use of secure enclaves, namely, Intel SGX.

1. Introduction

Most applications assume that the environment they are running is trustworthy and thus does not employ any defense mechanisms to secure sensitive data (e.g., cleartext storage of private keys and passwords). This is because unprivileged software alone cannot defend its memory contents from a more privileged code, i.e., the kernel without extra protection primitives provided by the hardware.

At first glance, it might not be obvious to consider the operating system or even the hypervisor in the threat model on devices that we control. However, a malicious hypervisor can modify and read any physical memory present in the machine, whereas a malicious kernel can do the same to any memory allocated to user processes.

The issue above can be partially solved with homomorphic encryption, a cryptography field that allows a computer to perform operations on the encrypted data as if it were performing on the original data. This means that the decrypted result after the operation must be equal to the computed value if performed on the original data. Unfortunately, not many implementations exist that apply homomorphic encryption are symmetric, and those that are, most were broken (Fontaine and Galand, 2007). Most applications are asymmetric, which suffer from the performance penalty relative to their symmetric counterpart (Fontaine and Galand, 2007).

A more feasible alternative solution can be achieved through the use of secure hardware components. Some ex-

[*] Corresponding author.
E-mail addresses: andre.brandao@fc.up.pt (A. Brandão), jresende@fc.up.pt (J.S. Resende), rmartins@fc.up.pt (R. Martins).

https://doi.org/10.1016/j.cose.2021.102327

amples include secure elements (SEs), Trusted Platform Modules (TPMs), Java Cards, Intel Trusted Execution Technology (TXT), Trusted Execution Environments (TEE) and Secure Encrypted Virtualization (SEV). Some with more limitations than others, but all generally provide a way to secure computation. The major limitation of these technologies is that they require dedicated hardware in the host machine.

The recent advancement in the trusted execution environment (TEE) technology (e.g., Arm TrustZone, Intel SGX and AMD SEV) has allowed today's common off-the-shelf (COTS) CPUs to create such programmable environments on enterprise and consumer-grade devices. This easily enables anyone to work on such environments and quickly deploy it in the real world, thus allowing applications to consider a ”wider” threat model, as it is, in theory, protected against privileged code, i.e., running below security ring 3.

1.1. *Motivation*

Today's general-purpose computing devices run all sorts of software, each one of them with their potential set of vulnerabilities. This means that compromising one of them may lead to access to the device, potentially compromising the remaining software.

Intel SGX gives the developers stronger guarantees that the code executed was not tampered with while providing confidentially of the data and code inside it. At the same time, it provides the opportunity to securely save sensitive data on external devices through a per-device and per-application encryption key. This also means that even though with the increasing popularity of the cloud and many companies moving their infrastructure to uncontrolled hardware, they can securely utilize it, without the fear of leaking sensitive data to unknown third parties.

Given that porting an application to a new environment is usually hard, not much work has been done to attempt running day-to-day server software on Intel SGX. Focus has mainly been on tools that separate code automatically (Lind et al., 2017) or running unmodified applications inside an enclave (Arnautov et al., 2016; Baumann et al., 2015; Priebe et al., 2019; Tsai et al., 2017). The main issue associated with these approaches is that they do not consider each application's specificity leading to incompatibilities or/and significant performance loss.

In this work we aim to mimic an hardware security module through an Intel SGX enclave, giving it similar guarantees as an Hardware Security Modules without the inconvenience of requiring the deployment of additional hardware on the host.

1.2. *Contributions*

The main goal of this work is to study the feasibility of using secure enclaves in real-world scenarios to provide integrity and confidentiality of both data and computation. In summary, the contributions of this work include:

- Identify existing server applications used in the real world and solutions that can leverage Intel SGX;
- Applicability of Intel SGX in applications used in the real world;
- Open source implementation of prototypes and corresponding deployment utilizing Intel SGX;
- Definition of a threat model with security analysis for each scenario;

1.3. *Outline*

We start by presenting the motivation for this work and the contributions achieved in this work. This is followed by Section 2, where we introduce some concepts necessary throughout this work. It gives a brief overview of how modern processors protect higher privilege codes and the multiple ways to achieve virtualization in the modern world. This Section also explained what a trusted execution environment is and how it can break the typical security ring in modern processors.

In Section 3, we present related work that aims to run applications in Intel SGX, summarize the contributions of other similar work and discuss limitations of existing approaches.

Section 4 presents our work on Apache's web server, the creation of an OpenSSL Engine, both applied to Intel SGX. Additionally, with each implementation, we perform a security analysis.

In Section 5, we define the test methodology to evaluate our implementations and present the results so that we can compare them to their original counterpart and, when applicable, to other existing solutions.

Lastly, Section 7, concludes this work by over-viewing the results obtained in the previous Section. Besides, we discuss a set of limitations of the current implementations and possible future iterations of this work.

2. Background

In this Section, we address some necessary concepts before tackling the problem. We start by describing the concept of security rings in modern processors, followed by an explanation of current virtualization techniques. For last, we introduce some basic concepts of Intel SGX that will be necessary throughout this work presentation.

2.1. *Security Rings*

Modern processors typically have several execution modes that provide hierarchical layers of privilege, known as the security rings (Guide, 2011). The lower the ring in which the CPU mode operates, the higher the privilege is. Even though x86 gives various security rings to work with, general-purpose operating systems such as Linux and Windows only leverage two CPU modes, running code in kernel mode, ring 0, and user mode (ring 3).

With the evolution of the architecture and the introduction of hypervisors, deeper levels of privilege were introduced. One example of this is the hypervisor mode, also known as ring -1, capable of preempting and isolating kernel code. Ring -2 refers to the system management mode (SMM), which can seize the hypervisor code and has nearly unrestricted access to the system (Domas, 2015). It is also in charge of controlling power management, system hardware and run proprietary code from Intel and the motherboard manufacturer.

Trusted execution environments may break the hierarchical layers of privilege by only allowing software to run in a less privileged code and denying access to higher privileged code.

2.2. *Trusted execution environment*

Trusted Execution Environment (TEE) is often referred to as a secure, integrity-protected programmable environment with memory and sometimes storage capabilities (Sabt et al., 2015).

Global Platform defines a ”TEE system architecture” (GlobalPlatform, 2017) which systems must comply in order to be considered a Trusted Execution Environment. At a very high level, the requirements for these are:

- Protect assets from environments other than the TEE itself.
- Protection against some physical attacks
- System components (e.g., Debug interfaces) capable of assessing TEE assets are disabled or controlled by an element protected by the TEE.
- The TEE must be instantiated through a ”Secure boot” process by the SoC or an Off-SoC Security processor
- Provide trusted storage of data and keys
- Software running outside the TEE should not be able to call internal TEE APIs directly

We can see how a TEE may break the hierarchy of security rings from the first and last requirements. Any code other than the one in the TEE itself should not be able to access the TEE contents. This includes code running in higher privilege modes, e.g., kernel code.

2.2.1. *Intel SGX*

In 2015, Intel introduced, along with the Skylake microarchitecture, Software Guard Extension (SGX), a set of security instructions that aims to provide users with a hardware implementation of a Trusted Execution Environments(TEE), allowing integrity and confidentiality guarantees to computation performed on a device even if all privileged code is compromised.

The creation of a trusted execution environment in Intel SGX is achieved by allocating processor reserved memory (PRM), which the processor protects from all non-enclave memory accesses, including from kernel, hypervisor and system management mode code (Costan and Devadas, 2016).

Memory structure

The PRM holds Enclave Page Cache (EPC) sets, each with 4 KB of size, which are assigned to the enclaves by untrusted software. The CPU makes sure that each EPC belongs exclusively to one enclave by maintaining an Enclave Page cache Metadata.

As the processor reserved memory for Intel SGX is limited to a maximum of 128MB (Intel, 2011), Intel SGX provides instructions for the Operating System to evict EPC pages to untrusted memory and later load them back. As the memory where the evicted EPC pages are stored is readable by the operating system, SGX uses cryptography operations to ensure the integrity, confidentiality, and freshness of the evicted EPC pages (Costan and Devadas, 2016). As the EPC pages and other SGX specific data are required to be stored in the PRM, the usable memory for applications within Intel SGX is limited to approximately 90 MB.

Threat model

Intel SGX’s threat model assumes that the operating system and all application code could be compromised or malicious and are considered untrusted. The CPU guarantees that the enclave memory can only be accessed from the code running inside the enclave itself. This allows for enclaves to execute sensitive computations without worrying about malicious privileged code to read the sensitive data.

Intel SGX does not protect against application bugs (Lee et al., 2017a; Weichbrodt et al., 2016) within the enclave, bugs on the implementation of Intel SGX, nor does it guarantee safety against side channels attacks.

Memory safety violations

Enclaves can leverage code secrecy by self-modification during runtime (Bauman et al., 2018; Schuster et al., 2015). This poses a problem for those wanting to leverage return-oriented programming (ROP) to exploit existing software. By monitoring the exceptions thrown by the enclave, Lee et al. (2017b) demonstrate new techniques to find buffer overflows, ROP gadgets, and the desired functions (e.g., memcpy) in enclaves utilizing code secrecy. This allows the attacker to build an ROP-chain to memcpy to copy data from the enclave to normal memory.

Side channel attacks

A side-channel attack aims to obtain information about an executing system through information leaked through a side channel, such as power consumption, timing information, or even sound. Currently, all known vulnerabilities affecting Intel SGX can be mitigated through microcode updates from Intel.

Prime+Probe is an example of a side-channel attack, and it leverages the L1 cache of the processor to determine what addresses were accessed. First, the attacker primes the cache by accessing memory to fill the L1 cache in its entirety. Afterward, when the victim’s process accesses memory addresses, some previous L1 cache portions are evicted and loaded with the victim’s data. The attacker can now probe the same addresses and measure the time it took to access each address since accesses to the L1 cache are faster than the ones to ram. He is now aware of which cache lines got evicted. As the attacker knows the code and the victim’s accesses pattern, he could potentially extrapolate information about sensitive data.

Brasser et al. (2017) demonstrate the Prime+Probe attack applied to Intel SGX. Firstly, it requires that the enclave code is executed in the same core as the attacker’s process, requiring modifications of the scheduler. Secondly, the enclave’s uninterrupted execution is necessary so that the L1 cache is not polluted further, making it a requirement to have simultaneous multi-threading (SMT) enabled, known as Hyper-Threading on Intel processors. The last condition also leads to the kernel’s necessity never interrupting the core on which both the victim’s code and attacker’s code run.

Van Bulck et al. (2018) exploits speculative attacks to read memory from Intel SGX protected memory regions. This includes the secrets used to seal data and pass attestation services. Due to SGX's privacy features, an attestation report cannot be linked to its signer's identity. This means a single compromised SGX machine could erode trust in the entire SGX ecosystem. To fix this, Intel issued an update to the microcode of the affected CPUs and revoked the attestation keys extracted with by foreshadow.

More recently, in Minkin et al. (2019) was demonstrated an issue in an undocumented optimization within Intel CPUs, which was named Write Transient Forwarding (WTF). When an instruction attempts to write a value to memory, the processor needs to translate the virtual address to a physical address to acquire exclusive access to it. To prevent stalling the store instruction, the WTF optimization stores the address and value in a buffer and continues executing the program. Later, the addresses in the buffer are resolved and used to store the values in memory. Once a value is stored in the buffer, subsequent loads to that address need to load the buffer's value so that stale values are not read from memory. The processor matches the address in the load instruction to the ones stored in the buffer.

To make the decision faster of where the values are stored in the buffer, partial address matches are used to rule out the need for store-to-load forwarding. An issue arises when a load instruction with an address stored in the buffer that is bound to fail (e.g., through an access violation) incorrectly forwards the value of the partially matched store instead of cleaning the CPU state. An attacker may generate faults so that load instructions would cause an error and incorrectly forward the store value. Subsequently, an attacker can use a Flush+Reload side-channel attack similar to Prime+Probe to read the forwarded value. Intel classified this issue as a Microarchitectural Store Buffer Data Sampling (MSBDS).

Schwarz et al. (2019) demonstrates the issues of MSBDS in real-world scenarios leaking data from user-space applications, the kernel, Intel SGX enclaves, other virtual machines, and even the hypervisor. The same paper showed a new technique similar to MSBDS that, in addition to Intel TSX, allows for the data leakage to occur on Intel Cascade CPUs, supposedly resistant to MSBDS.

Cache-out van Schaik et al. (2020b) demonstrates that Intel's fix on Whiskey Lake CPUs is not enough to mitigate MSBDS attacks. It also demonstrated that an attacker could select which cache sets to read from the CPU's L1 cache. Additionally, because the L1 cache is often not cleared on context switches, it is feasible to exploit even on CPUs with Hyper-Threading disabled, where the victim's code runs subsequently to the attacker's code. This attack can extract secrets in Intel SGX enclaves van Schaik et al. (2020a), including the keys used to seal data and pass attestation. This essentially allows for any code to pass as a legitimate enclave, even if not running within Intel SGX. Like foreshadow, this requires for the extracted keys to be revoked by Intel.

2.2.2. *OpenEnclave*

OpenEnclave (OpenEnclave, 2020) is a hardware-agnostic Software Development Kit (SDK) for trusted execution environments, currently only supporting Intel SGX and Arm TrustZone. This allows any developer to support a multitude of enclave solutions without worrying about each system's specifics. It automatically partitions applications into two components, one to be executed inside the enclave and the other outside the enclave for operations not permitted inside the enclave (e.g., system calls).

3. Related work

In this Section, we will learn about some applications that Intel SGX has had throughout its life by discussing their key features and comparing them when applicable by pointing out the advantages and disadvantages of each solution

Firstly, we explore technologies that aim to run unmodified applications within a secure enclave's limited instruction set. Secondly, we analyze some applications that have been created with Intel SGX in mind, which results in much smaller code sizes than the previous solutions.

3.1. *Running legacy application on Intel SGX*

Since porting entire applications is typically an arduous task, focus has primarily been on automatically porting applications (Lind et al., 2017) or running unmodified applications inside Intel SGX (Arnautov et al., 2016; Baumann et al., 2015; Priebe et al., 2019; Tsai et al., 2017). As the enclave in Intel SGX excludes the operating system from its trust computing base (TCB), thus eliminating the *syscall* instruction, it means the number of applications that can run natively, without any changes, on Intel SGX is somewhat limited. This Section approaches some solutions that aim to run unmodified applications inside the enclave and identify the key differences in each solution.

While state of the art solutions achieve the same objective of running unmodified applications inside Intel SGX, this can be achieved in various ways. Fig. 1 shows three possible ways that achieve similar results.

Fig. 1a shows a design that places a Library OS and a shielding mechanism inside the enclave. The library OS allows the enclave to drastically reduce the number of system calls made to the kernel, thus decreasing the performance penalty associated with leaving and entering the enclave. The shielding layer protects a security-sensitive set of system calls, by, for example, encrypting and decrypting I/O operations. The disadvantage of this type of design is that by integrating the library OS inside the enclave, we are significantly increased the size of TCB. Fig. 1b shows the extreme opposite of the previous design. The application and its libraries are loaded onto the enclave with a shim C library. The shim library intercepts the C library calls and redirects them to the C library that is loaded outside the trusted environment.

To our knowledge, no current implementation follows this extreme approach as they typically implement some shielding layer. This solution would also imply a significant increase in the number of transitions to and from the enclave, decreasing the application's overall performance.

Finally, Fig. 1c shows a system that gathers the best of both previous solutions. It includes in its TCB a C library along with a shielding layer. All system calls are passed down to the operating system either by the C library or the shielding layer.

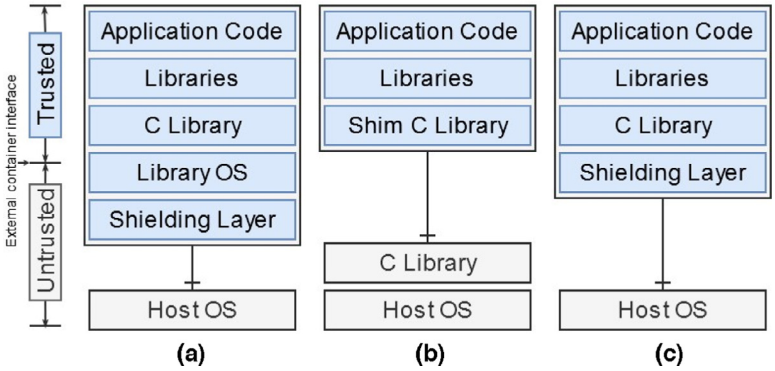

Fig. 1 – Secure containe designs Arnautov et al. (2016).

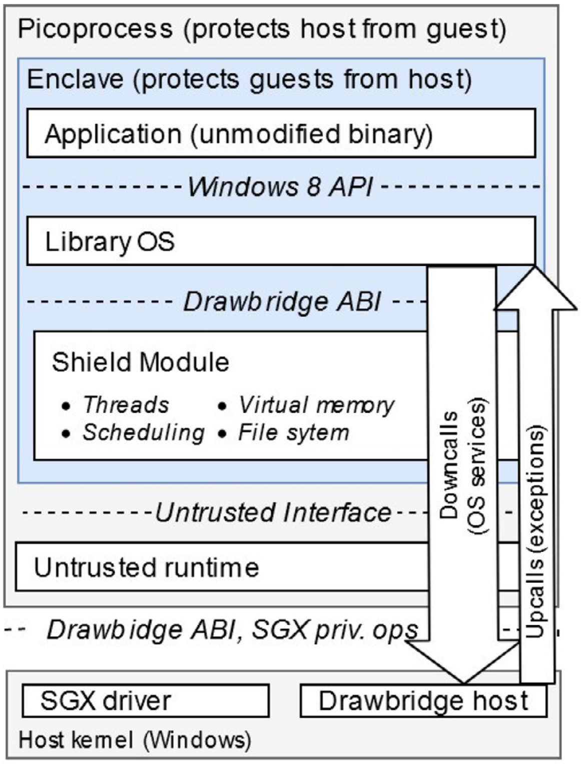

Fig. 2 – Haven components and interfaces Baumann et al. (2015).

3.1.1. *Haven*

Haven (Baumann et al., 2015) was the first solution to implement this kind of paradigm and follows the implementation in Fig. 1a. It allowed to run unmodified applications shielded from the operating system. To achieve this, Haven builds on top of Drawbridge (Porter et al., 2011) a system supporting sandboxing of Windows applications leveraging two mechanisms, pico processes and a library OS as seen in Fig. 2.

A pico process can be seen as a container similar to a docker container. It provides a relatively narrow application binary interface (ABI) consisting of downcalls, requests for OS services and upcalls, utilized for initialization, thread startup, and exception delivery. The pico process is also a way for the operating system to defend itself from the guest (the application).

The job of the library OS in Haven, besides providing a "user-mode kernel", is to provide an abstraction layer to the application with the ABI and the shielding layer. Because Intel SGX protects the enclave from the remaining system, Haven enables a mutual distrust between the guest and the host.

3.1.2. *Scone*

Scone (Arnautov et al., 2016) is a Linux solution similar to Haven, with the exception that it does not require a library OS to be loaded to the TCB (Fig. 1c). To support applications inside the enclave, it runs inside the TCB, a modified musl C library (musl, 2019). This solution also provides a M:N threading model in which application threads inside the enclave are mapped to N OS threads.

Due to the lack of a library OS, Scone relies heavily on the operating system to handle all the system calls. While on applications with a low amount of system calls, performance does not suffer too much, on applications with a higher rate of system calls, performance degrades significantly. To combat this issue, Scone allows users to load an additional kernel module to enable the enclave to perform asynchronous system calls.

As all system calls can potentially return malicious values, the shield layer must perform verification, similar to how the kernel OS protects itself from data coming from userspace.

Scone also permits shielding of external interfaces with transparent encryption of files, transparent encryption of communication channels via transport layer security (TLS) and transparent encryption of console streams (STDOUT, STDERR and STDIN).

Scone also allows us to keep confidentiality and integrity on files written and read by the enclave.

This solution has, since its release, gone closed source offering both a community edition and paid services (Scone,). The community edition runs exclusively in pre-release mode with debugging enabled and provides a set of curated images ready to be used. Alternatively, the community edition also provides compilers and runtimes for some languages. In the original paper (Arnautov et al., 2016) an optional kernel module is mentioned to improve system calls performance, but the website (Scone,) for Scone does not mention this feature.

3.1.3. SGX-LKL

SGX-LKL (Priebe et al., 2019) is a fork of the early stages of Scone, and contains some similarities. Like Scone it provides similar functionalities, such as transparent file encryption and a M:N threading model. Contrary to Scone it embeds a library OS, the Linux Kernel Library (LKL), in the TCB.

As it contains a library OS inside the TCB, it allows for a minimal host interface, providing only seven system calls. One system call for time-aware applications, two system calls for disk I/O, two system calls for network I/O, and two system calls for signaling.

SGX-LKL does not provide the same type of shielding as Scone for network connections, but it allows the enclave to create a TAP device connecting to a network via the wireguard (Donenfeld, 2017) virtual private network (VPN). This essentially allows the creation of a distributed network guaranteeing that only trusted nodes are present.

To hide disk I/O events instead of instantly performing a disk I/O call, these are inserted into a queue until there are enough changes to be written to disk. This allows us to hide from the operating system when the applications are reading or writing specific files. If the application does not generate enough disk I/O activities, random activities are performed to mask it. This prevents side-channel attacks on the enclave from operating regarding the enclave's inner workings when working with I/O operations.

3.1.4. Graphene

Graphane (Tsai et al., 2014) is a library OS that aims to be as host independent as possible, aiming to be a substitute to current virtualization by containerization solutions. Its goal is to allow Linux applications to be executed in any environment (e.g., BSD, OS X, Windows), without relying on virtualization. This is achieved by creating a similar architecture to Haven, a pico process is created with the wanted executable, its dependencies and the Graphene library OS. System calls are translated to the host via a platform abstraction layer (PAL), which, as the name implies, is an abstraction layer that changes with the operating system. One key difference from Graphene to Scone and its derivative works is that instead of using the musl c library, it uses the gnu c library.

Graphene-SGX (Tsai et al., 2017) handles Intel SGX as just one more environment on which the applications may run. To run unmodified applications under Intel SGX, a port of the PAL was made to Intel SGX. This allows, similarly to Haven, a mutual distrust from the host and the guest. It is one of the few, if not the only solution, that allows the creation of forks of existing processes. When a process is forked, a new clean process is created. Then the two enclaves, via an inter-enclave remote process communication stream, exchange an encryption key, validates the CPU-generated attestation of each other, and migrates the parent process snapshot. The current solution leaves for future work the protection of the network and file system.

To load an unmodified program to Graphene-SGX, it first creates hashes of the program, its dependencies, saves it to a file, and signs it, essentially creating a whitelist of permitted files. Later, when the application executes inside the enclave, checks the executable and the utilized files against the created whitelist.

3.2. SGX native applications

In this Section, we overview some solutions that utilize Intel SGX in order to widen its threat model to have stronger guarantees that the desired data is kept confidential from even the operating system itself.

3.2.1. SafeKeeper

SafeKeeper (Krawiecka et al., 2018) is an approach to protect the confidentiality of passwords in web authenticated systems through the use of Intel SGX, protecting even from malicious or compromised servers. It considers a very strong adversary in its design, with access to the password database, ability to modify the web content sent to the user, access to the server-client communication, server-side code execution and phishing attempts. It also assumes that the user only enters passwords on SafeKeeper enabled web services.

This approach has two components, a server-side password protection service to safeguard the passwords from the rest of the code running in the server and a browser add-on to correctly identify web services running SafeKeeper and securely communicate with them.

The server-sided password protection service is designed as a drop-in replacement for existing one-way functions. It takes a salted password as an input and the result is a keyed one-way function, which is stored in the database. In order to protect the key utilized, it never leaves the enclave in clear text, all computation of the one-way function is made inside the secure enclave. An adversary with access to the password database cannot perform an offline password-guessing attack as it does not know the key used in the one-way function. This forces the adversary to an online-only type of attack against the web service, which can be rate limited by the service.

The browser add-on needs to correctly identify that it is communicating with the SafeKeeper password protection service to transmit the user's password securely. The identification of the service is made via remote attestation of the secure enclave to guarantee that it is talking to a genuine Intel SGX enclave and to verify its contents. Additionally, this attestation protocol establishes a shared session key in order for the client to establish a secure communication channel with the enclave, on which the login credentials will be sent.

3.2.2. Intel SGX Key Store

Keys in the Clouds is a solution presented by Kurnikov et al. (2018) that aims to create a web service that utilizes Intel SGX as its trusted execution environment to create a secure key store accessible from anywhere.

It works as a web service and leverages Intel SGX to store and utilize the key securely, the service permits the key owners to utilize keys, delegate it to other users and audit its usage. Integration with GnuPG and OpenKeychain are provided on Android systems.

To guarantee that the user is in fact, speaking with a genuine Intel SGX enclave, remote attestation is performed through the same efficient remote attestation protocol as proposed by the SafeKeeper.

3.2.3. *tpmsgx*

The utilization of Intel SGX on a cloud-like environment is especially difficult because data is encrypted with a per enclave per device key and because the physical memory available to the program is at most 128 MB, with around 40 MB being already used for the management of SGX itself.

Tian et al. (2019) propose a system so that applications can leverage Intel SGX in a cloud-like environment for multiple users through the use of an emulated TPM and LXC containers.

The solution acts like a ”TPM as a service” for applications to use without resorting to solutions like Haven (Baumann et al., 2015) that requiring moving a big code base to the enclave. Unlike traditional TPMs, it does not allow the TPM use before the operating system is initialized, meaning it cannot be used for secure boot. This service allows for multiple applications to utilize a single enclave as TPM, rather than each application initializing its own enclave.

The implementation specifies that, if remote attestation is enabled, communication between the server and the client is encrypted by an AES128 shared secret that is established between them. The publication does not specify if or how the client authenticates communicating with a genuine Intel SGX enclave.

3.2.4. *SGX-Kernel*

Although Intel SGX is limited to running code in user mode, Richter et al. (2016) propose a solution that allows the kernel to delegate some of the work to an enclave, in order to isolate kernel with Intel SGX.

The system is compromised by two components, a kernel module, and a secure enclave running in user mode. The kernel module acts as proxy for the secure enclave and their communication is made through a Netlink interface, allowing the kernel to delegate work to be done in the secure enclave.

A proof of concept is demonstrated by creating a file system managed by the enclave, which is responsible not only for its storage but encryption as well, guaranteeing, that the encryption key is never exposed to other applications or other kernel mode code.

3.2.5. *Aurora*

Secure enclaves within Intel SGX are limited to a subset of instruction, this coupled with it running exclusively in user mode, disallows it from communicating directly with the hardware, because of this I/O is typically left out from the threat model. AURORA (Liang et al., 2019) aims to solve this by creating a trusted path between an enclave and a target I/O device. This is achieved by adding code to the Unified Extensible Firmware Interface (UEFI) to reroute I/O interrupts to the System Management Interrupt (SMI) by configuring the Advanced Programmable Interrupt Controller (APIC) and registering SMI handlers. SMVisor is the core component in AURORA, that handles this notifies the enclaves as necessary via an inter-processor interrupt. In order for the enclave to communicate with SMVisor a special ioctl is utilized on the ashmd module, a module which allocates contiguous physical memory as shared memory in order to facilitate the communication between the enclave and the SMVisor.

To guarantee that both the SMVisor and the enclave are not communicating with a malicious party or to guarantee that no man in the middle attack is occurring, mutual attestation is performed. During a measured boot, Intel Boot Guard hashes the firmware and saves the result in the TPM Platform Configuration Register (PCR). An enclave can confirm that SMVisor has not been tampered by querying the TPM which will answer with a cryptographic signature over the PCR value along with a nonce to prevent replay attacks. The attestation of the enclave is performed during the launch of the enclave via Intel’s local attestation.

4. Design and implementation

This Section describes the designs and implementations for tamper prevention of sensible data, particularly encryption keys. First, we show a modification to Apache’s web server where the TLS termination is moved to a secure enclave. Subsequently, in Section 4.2 we show an encryption key storage implemented in Intel SGX leveraging OpenSSL engines. This solution offers less protection than the former but is contrasted by its versatility and better performance, allowing any application to integrate with it through OpenSSL easily.

4.1. *Integration of Apache’s web server SSL module within Intel SGX*

This Section describes the modifications made to mod_ssl to make it compatible with Intel SGX to protect both the asymmetric private key and the negotiated symmetric key from unwanted third parties. This was achieved by moving the TLS implementation to the TEE, and thus preventing access from both the operating system and the hypervisor. It also analyzes various cryptography libraries available to Intel SGX and identifies that WolfSSL (WolfSSL,) is suitable for our needs and identifies a new issue on the TaLoS (Aublin et al., 2017) library.

The implementation described in this Subsection is open source and available in a public repository on GitHub (Brandão, 2020a).

4.1.1. *Architecture*

Our system should follow the typical architecture of any Intel SGX application containing both untrusted and trusted components. The former is responsible for the interaction with the operating system interacting and querying the latter when working with sensitive data when necessary. This segmentation of the application is necessary due to the limited instructed set an enclave has access to, which does not allow it to communicate with the operating system via the *syscall* instruction.

Fig. 3 demonstrates how the modified SSL module works at a very high level. By redirecting the calls from the original OpenSSL to our enclave (1 in Fig. 3), the enclave can utilize the private keys or other sensitive data in its possession along with a cryptography library (2 in Fig. 3) to establish a TLS connection so that it can encrypt the outgoing data and decrypt the incoming data.

This guarantees us that the asymmetric private key and the randomly chosen symmetric key of the TLS connection

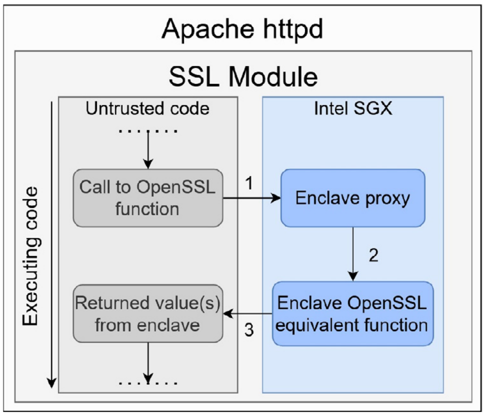

Fig. 3 – Proposed architecture for the Apache's HTTP server.

stay inside the enclave, making it impossible to be read by unwanted parties. Finally, as with any application, the enclave must return the result of the function to the untrusted code (3 in Fig. 3) so that normal operation can continue. If the resulting value to the function call to the cryptography library is a pointer, a random id is generated, added to the proxy and returned instead of the pointer.

To mitigate arbitrary reads and writes to the enclave's memory (Cloosters et al., 2020), we must implement a proxy that sanitizes the inputs received by the untrusted application seen in Fig. 3.

4.1.2. *Cryptography library*

Intel provides a fork of OpenSSL compatible with Intel SGX (Intel, 2017)(Intel SGX SSL), but its functionality is rather limited, not allowing for an application to terminate its TLS connections inside the enclave.

Our second choice was TaLoS (Aublin et al., 2017), a LibreSSL implementation for Intel SGX, which allows applications with little to no changes to terminate their TLS connection inside the enclave. This would be the ideal candidate since LibreSSL itself is a fork of OpenSSL and would require minimal changes to achieve our goal. However, Cloosters et al. (2020) discovered several security vulnerabilities that allow arbitrary read and write to the enclave from the untrusted code. The patches necessary to fix this issue are substantial, and the author of the library claims it will not be fixed. We will, later, explain how we prevent such attacks in our solution.

The chosen library for our solution was WolfSSL (WolfSSL,), which supports terminating TLS connections inside Intel SGX. Additionally, it also contains a compatibility layer for OpenSSL, allowing it to be used on most applications that utilize OpenSSL.

4.1.3. *Mitigating memory corruption vulnerabilities in SGX enclaves*

Although there are legitimate cases to receive and return pointers without safety checks, this is highly discouraged. If incorrectly used, it could very easily allow memory corruption within the enclave from untrusted code, leading to arbitrary reads, writes and maybe even code execution in certain cases (Cloosters et al., 2020).

Any function within the enclave that accepts a pointer without safety checks and is exposed to untrusted code could be a potential read/write primitive for an attacker.

A.1 shows a very simple function in WolfSSL (WolfSSL,). It returns the variable *rfd* in the pointer to a structure of type *WOLFSSL*. At a lower level, this means that the processor will add the offset of *rfd* in the structure *WOLFSSL* to the pointer provided in *ssl*, read the contents of the resulting address, and return it. While it may seem harmless to expose this function to untrusted code, this gives an attacker a read primitive to an Intel SGX enclave. This is because nothing guarantees that the pointers passed to the function are of the type *WOLFSSL*. This function will return an integer at the offset of variable the *rfd* in the *WOLFSSL* structure pointed by the parameter *ssl*. Since the parameter *ssl* can point to anywhere in memory, an attacker can read any memory inside the enclave. Similarly, the function in A.2 shows a write primitive to an Intel SGX enclave if exposed to untrusted code.

To mitigate this attack, the enclave must perform safety checks on the passing points. In our solution, we chose to create a hash map for each type of structure used and exported to untrusted code. This allowed for constant-time access to the saved pointers regardless of the number of existing pointers.

When the untrusted code calls a function in the enclave (1 in Fig. 4), the enclave will first check if any of the ids passed exist in the hash map. If the passed ids contain a valid entry in the hash map, the call is forwarded to WolfSSL (2 in Fig. 4). If the resulting function returns a non-NULL pointer, it is checked if it exists in the hash map and returns the corresponding id to the untrusted application. If no entry is found, it will generate a truly 64-bit integer, through the *rdrand* instruction, insert it in the hash map with the corresponding pointer, and return the id to the untrusted code (4 in 4).

A hash set could have been used in place of the hash map, but it would require returning a pointer inside the enclave to untrusted code. As any data that resides within the enclave is not accessible, we believe that this information is not important to the untrusted code. This way, we prevent the attacker from gaining knowledge about the location of the objects inside the enclave. We also believe that returning an id instead of the pointer to the enclave's data may hinder heap-spraying. This attack consists of allocating large amounts of memory in the heap so that an attacker can place the desired data in a predetermined location. Because allocations will always return a random 64-bit integer, the attacker will never know if data was placed in the desired location.

Having a single hash map would probably suffice in translating ids to pointer (i.e., ids to *void**). We decided to create a hash map for each type used within WolfSSl/OpenSSL. This allowed even for adequate control over the ids passed as it guarantees that the wrong object type is never passed to a function (e.g., passing a WolfSSL_RSA pointer to a function that expects a pointer to BIO), without a performance penalty over using only one hash map.

When the function to free the object is called, the corresponding id and pointers are removed from the corresponding hash map. This means that the untrusted code cannot intentionally leverage use-after-free exploits as when calling a function with an id that is no longer in the hash map will

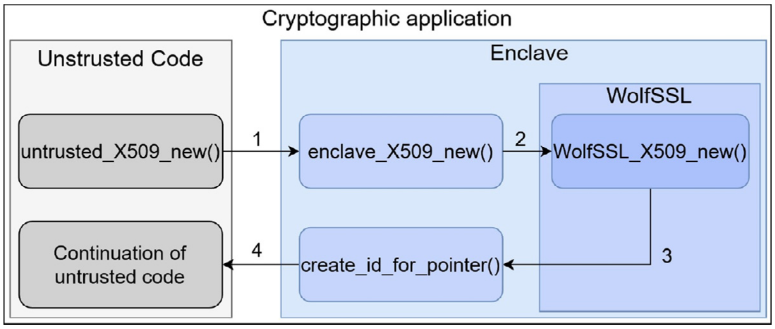

Fig. 4 – Preventing memory corruption in the enclave.

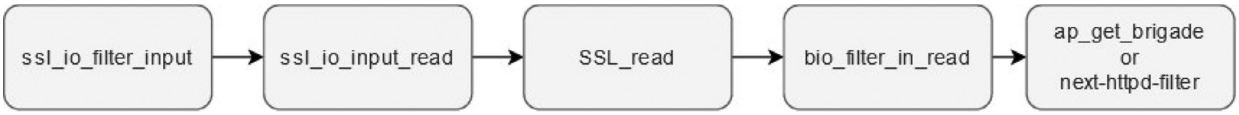

Fig. 5 – mod_ssl input chain

cause the function call to not be forwarded to WolfSSL and instead return with an error.

4.1.4. *Changes to mod_ssl*

The approach taken to modify mod_ssl was to keep the original code untouched as possible by changing only OpenSSL functions' calls. This would allow us to merge eventual patches from the original branch more easily. Unfortunately, since the cryptography library's memory region is not accessible from the module, some issues arise, requiring modifications to the code base of *mod_ssl*.

OpenSSL callback support

The original TLS module takes advantage of OpenSSL's input/output stream abstraction layer to tell OpenSSL how it should read data for the *SSL_read* and *SSL_write* functions. An application can register a callback that will be called by OpenSSL to write and read the encrypted contents.

After the connection is upgraded to TLS, it creates an SSL context for the connection and sets the callbacks to handle the input/output.

Fig. 5 shows roughly the input chain used in mod_ssl. First the http server notifies the module via the *ssl_io_filter_input* a request for data. If it determines that it should read data, it will call an helper function which calls the OpenSSL function *SSL_read*. Since OpenSSL has been instructed to use *bio_filter_in_read* to read the data from, it will call it. Finally *bio_filter_in_read* will then try to fetch data from the next httpd filter and return it to *SSL_read*, which will decrypt it and return it to *ssl_io_input_read*. The output chain follows the same pattern, registering a function to tell OpenSSL on how to perform writes

If the cryptography library is inside an Intel SGX enclave, the application may still register the callback to functions outside the enclave, but when the enclave tries to execute that function, a segmentation fault will occur. This is because enclaves are not allowed to execute instructions outside its memory range without first issuing a special *ocall* instruction to leave the enclave.

To overcome the issue mentioned above, we resort once again to hash maps, which will hold a pointer to an array containing all possible callbacks in the *BIO_METHOD* and a static callback for each callback which will resolve the correct pointer and forward it to untrusted code to execute. B.1 shows a simplified concept of this in practice, when *BIO_meth_set_read* is called instead of forwarding the call to the enclave, it is saved in its corresponding slot in the created array, the registered callback to WolfSSL will instead be a function within the enclave. When called, the callback will obtain the real pointer saved in the array and tell the untrusted code to execute it. The functions *GetBioCallbackArray* and *CreateBioCallbackArray* simply get and create the necessary array in the hash map.

Additional getters/setters

Even though OpenSSL 1.1 removed many of its structures from the public header files so that they become opaque and force the usage of accessor functions (OpenSSL, 2017), not everything got the same treatment. Structures such as the *GENERAL_NAME* are not opaque and still rely on the application to directly access the data in the pointer returned by OpenSSL. This poses two problems, one specific to our solution and the other due to Intel SGX. First, as mentioned in Section 4.1.3, our library does not return real pointers, but instead randomly generated ids, meaning any attempt to dereference the pointer will most likely result in an access violation, crashing the program.

Secondly, Intel SGX enclaves when running in release mode do not allow access from untrusted code. This means, similarly to the first issue, trying to access the data it points to will cause an access violation. Because of the latter issue, we believe that TaLoS Aublin et al. (2017), even though it claims support for Apache's HTTP server, it does not handle these cases within the LibreSSL library. This means it will only work in pre-production mode, allowing access to the enclave's memory from any code. Attempting to use the library in production, i.e., enclave compiled in release mode, would cause segmentation faults when accessing data that resides in the enclave.

Sealed key detection

As we wanted to retain as many features as possible from the original module, we decided to load normal keys and sealed keys by the enclave. We modified the module first to load a sealed key, and if it failed, try to load the key normally. For convenience and help users identify what kind of key they have in storage, we append the suffix '.sealed' to the end of the file's name.

4.1.5. *Limitations*

During its normal operation, the Apache HTTP server will create multiple forks to handle multiple connections. However, this feature is not well supported by secure enclaves. Since the enclave's memory space is not accessible to the operating system, it cannot be copied to the forked process. While it would be possible to detect a fork and reinitialize the enclave, all its contents would be reset to the default values, meaning all the data that the server relied on are gone. Synchronization might be possible, but it would require significant changes to WolfSSL. This means our solution is limited to working in a single process, severely limiting the number of possible connections that can be handled simultaneously. This is a limitation present in TaLoS (Aublin et al., 2017) as their source code repository suggests running the HTTP server in single-process mode. To our knowledge, the only solution that implements and supports forking is Graphene-SGX (Tsai et al., 2017) through inter-enclave communication but with a high overhead when compared to the native fork of a normal application, with a latency increase of 8000 fold, which becomes worse with the increasing size of the forked enclave.

4.1.6. *Security analysis*

In this Subsection, we present a security analysis of the proposed solution to move the TLS termination to a secure enclave.

Access to encryption keys

Since the cryptography library's code is exclusively inside a secure enclave, this means that both the asymmetric private key and the negotiated symmetric key between the client and the server during the TLS connection handshake are kept secret by Intel SGX. Any external code to the enclave that attempts to access the protected memory regions will raise an access violation exception regardless of its privilege.

4.1.7. *Architecture*

Enclave memory corruption

As mentioned in Cloosters et al. (2020) if an enclave exposes functions accepting arbitrary memory points without safety checks, there may exist functions facilitating read and write primates to the enclave's memory, defeating the purpose of an Intel SGX. This might be considered a software bug, which means it is not something Intel SGX should be protecting against. To ensure that arbitrary write and read are blocked, our solution performs safety checks as presented in Section 4.1.3 before forwarding the request to the cryptography library.

Key usage

The current solution allows for any code to call the enclave and utilize the private key. Future iterations of this solution could log to a remote server the usages of the asymmetric private key.

4.2. *OpenSSL engine integration*

As we will see in Section 5, the previous solution comes with some big caveats and a high-performance penalty. Thus, we started looking for other ways to protect the private key used by untrusted software. This Section describes our implementation of a key store that uses Intel SGX to keep its contents secrets and its integration with an OpenSSL engine. This solution it allows for any application that uses OpenSSL as its cryptography library to use the keys protected by an enclave, guaranteeing its integrity and confidentiality, this approach works very similarly to a hardware security module, a pkcs#11 device, an Android's phone keystore system (Google, 2020) or web-services like HashiCorps' vault (Hashicorp,). Unlike the previous implementation it does not protect anything besides the private key utilized, meaning that if an application utilizes the private key to establish established symmetric keys in their TLS connections these would be accessible to an attacker with read access to the address-space of the application holding these keys.

The solution described in this Subsection is open source and available in a public repository on GitHub (Brandão, 2020b).

As in this implementation the enclave works in a very similar manner to a pkcs#11 device. It exposes a limited number of functions limiting the attack surface. The exposed functions only need to allow untrusted code to load private keys and perform operations with it (i.e. Encryption and Decryption). Table 1 shows the functions exported by the enclave and their functionality. As the *RSA* object in OpenSSL needs to contain at least the modulus and the public key exponent, a function exists to export values out of the enclave.

To make it possible for applications that use OpenSSL to use our solution easily, we implemented an OpenSSL Engine, which at a very high level essentially instructs OpenSSL on how to load and utilize keys from a custom solution. Our engine implementation was based on the Android Open Source Project (Google, 2015) due to its simplicity. As the source code targets a version of OpenSSL before 1.1, with the OpenSSL Engine's help for pkcs#11 devices, libp11 (OpenSC, 2020), we updated its source code to target OpenSSL 1.1.1g.

The Engine registers a callback on OpenSSL for the event that loads a key. When this callback is called, it attempts to load the key. If the key is successfully loaded from the key store, custom methods are set for the encryption and decryption of data when using the key. These methods, when called, will forward the request to the enclave.

We chose not to load the enclave inside the OpenSSL engine, as this would mean the enclave would need to be loaded in the same process as the application using OpenSSL. This

Table 1 – Exported function by the enclave to perform cryptography operations on a private key.

Exported enclave function	Purpose
enclave_private_encrypt	Encrypts(Signs) data with the given private key
enclave_private_decrypt	Decrypts data with the given private key
enclave_rsa_get_n_e	Gets modulus and the public exponent of a key
enclave_rsa_load_key	Loads a key into the enclave
seal_data	Encrypts data with an unique key to the enclave, used to 'import' keys
gen_rsa_key	Generates an RSA key inside an enclave, seals it and returns it

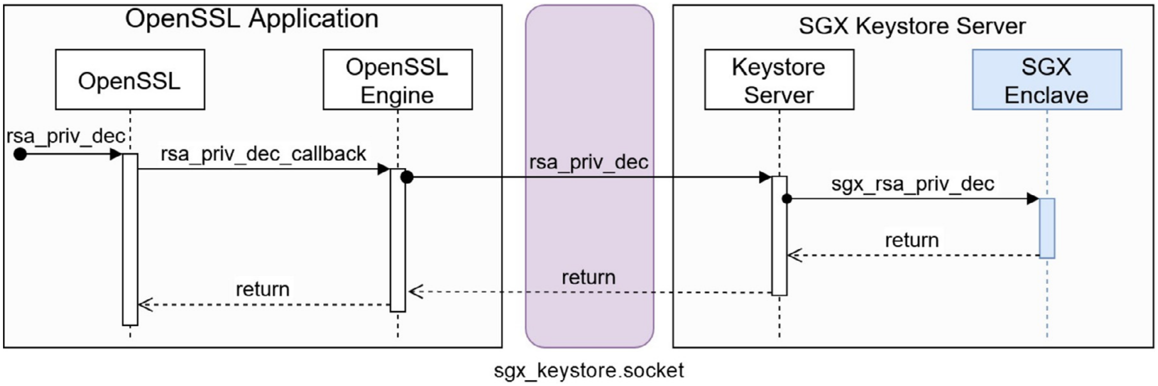

Fig. 6 – OpenSSL Engine with SGX key store architecture.

would require installing fork detection in our Engine and reinitializing the enclave in the forked process, ultimately wasting the physical memory allocated to SGX unnecessarily. Instead, our solution runs in a separate application, what we call the server, and listens on a UNIX socket domain for inter-process communication. The OpenSSL engine will then connect to the socket and forward its requests and wait for a reply.

In Fig. 6, we exemplify an application that utilizes a private key present within the enclave to decrypt some data. Firstly, OpenSSL receives this request and checks if any custom handlers for these requests exist, normally set by an OpenSSL Engine. When our OpenSSL engine receives a request, it attempts to establish a connection to the server and send the request to it. The server will then finally forward the request to the secure enclave, which will process the data and return it.

Similarly to the implementation in Section 4.1, to prevent memory corruption vulnerabilities within the enclave, no real pointers are passed to and from the enclave. When an application loads a key via the *enclave_rsa_load_key* function, a key id is returned and on the remaining functions that require a reference to the key, this id is used instead.

4.2.1. *Cryptography library*

Unlike the implementation in Section 4.1 the functionality required from the cryptography library is much smaller. The only requirements needed by the library is the ability to load an RSA private key and perform operation with it. Fortunately, even though the functionality of Intel SGX SSL (Intel, 2017) is rather limited, it provides support for the required features. We chose this library over WolfSSL (WolfSSL,) as it is a fork from OpenSSL. It is giving us a simple one to one match of the functions utilized by other OpenSSL Engines.

4.2.2. *Configuring OpenSSL*

To make OpenSSL aware of the Engine and make it so that other applications can use it, a few modifications need to be made to install OpenSSL. On an installation of OpenSSL through Ubuntu's package manager, B.2 needs to be added to the configuration file */etc/ssl/openssl.cnf*. In addition to configuring OpenSSL, the Engine needs to be copied to the configuration file's path. This should make it possible for any application to use the Engine, even from command like as shown in B.3.

4.2.3. *Required modifications to applications*

Applications that utilize OpenSSL may easily be adapted to use any engine available. By calling the OpenSSL function *ENGINE_by_id* with the id of the Engine, a reference to the Engine is obtained. Subsequently, the application must call *ENGINE_init* to initialize it. To load a private key from the Engine, OpenSSL provides a function *ENGINE_load_private_key*, which attempts to load a key from the Engine and returns a reference to a *EVP_PKEY* object which can be used as if it were a key loaded through the conventional OpenSSL API.

Adding support to Apache's HTTP server

When Apache's HTTP server attempts to load a key or a certificate, it checks if the provided Uniform Resource Identifier (URI) contains a colon and if its prefix is a supported OpenSSL Engine. If it is a supported engine, the server attempts to load and initialize the Engine with the same id as the prefix in the URI, in our case, sgxkeystore. For example, the URI *sgxkeystore:key.pem.sealed* would result in the initialization of the engine *sgxkeystore*, loading the key "key.pem.sealed". As the original code already supports pkcs#11 devices through libp11's

OpenSSL engine (OpenSC, 2020), adding support for another engine is a simple process. The only changes required are to detect the prefix in the URI, as shown by the patch in B.4.

4.2.4. *Security analysis*

This Subsection presents a security analysis of creating a key store with SGX and making it usable to external applications.

Access to Encryption Keys

Unlike the solution presented in Section 4.1 this implementation does not protect the symmetric keys during a TLS connection. It works very similarly to a hardware security module (HSM) or, more precisely softHSM (OpenDNSSEC,), which emulates an HSM in software. As our solution only implemented support for RSA keys, it only guarantees the secrecy of these keys. Meaning if the key store is utilized for TLS, the agreed symmetric key would be exposed to untrusted code.

Key Usage

Like the first solution presented in Section 4.1, the current implementation allows for any code being executed to make requests to the sgx key store to sign and decrypt data with the private key. The current solution utilizes UNIX domain sockets to communicate with other applications, this means that it is easily adaptable to network sockets so that future iterations could leverage a solution as presented in Kurnikov et al. (2018), allowing only authenticated access to the keys and logging key usage.

5. Results

This Section presents the performance tests made to our solutions and compares it to other existing solutions presented in Section 3.

Our evaluation used an Intel Nuc NUC6i7KYK, a quad-core 2.60 GHz Intel Core i7-6770HQ with 16GB of dual-channel DDR4 memory running Ubuntu 18.04.3 LTS, with Linux Kernel version 5.0.0-32. The amount of allocated memory to Intel SGX is 128MB.

The utilized version of Graphene-SGX was the one available in its public GitHub repository (Oscarlab, 2019) at commit *b4673dc171fbe4e972bea4dc79aae17212bc29da*.

Just like Graphene-SGX, SGX-LKL was obtained from its public GitHub repository (sgx, 2019) at commit *a4fc0 cc6fea39f30d33783e55626afbff3c7a871*.

To utilize Scone, access to the community edition was requested and granted. There wasn't any versioning besides the date on the docker images. The cross-compilers image from scone has digest *899ef9b2415bd2252c8a3ce396599 cc957405f9c9333f6b[0]7d39d95fe98fc00f2*.

5.1. *I/O Intensive application*

Not all applications can have the privilege to load all the necessary data to memory, especially in Intel SGX, since its physical memory is at most 128MB, with the application only being able to use around 90MB. To solve this, applications may load data as its needed. This implies that the application will need to make more I/O operations. As leaving and entering the enclave is a relatively expensive task, we believe it is also interesting to test the overhead in these types of applications.

5.1.1. *Methodology*

To test this type of load we have ten files, each containing 256 MiB of random data. To increase and decrease the amount of I/O operations, we change the amount of data that it is loaded to the enclave at a time. We utilize this data to calculate the sha-256 of the files with various buffer sizes to see how Intel SGX behaves with a large amount of transitions and exceeds the amount of physical memory available.

5.1.2. *Results*

In Fig. 7 Native is the application running normally without Intel SGX, it gives us a baseline so that we can compare with other solutions that leverage Intel SGX. We can see that increasing the buffer size beyond 1 MiB gives us diminishing returns.

Native-SGX represents our port of the same application to utilize Intel SGX, with a buffer of just 64 bytes. The average time taken to hash each file was 13.81 s. This first value of this solution is not represented in the graph because we chose to limit the graph's vertical axis to 5 s to get a better view of the other solution relative to each other. This considerable increase compared to the other solutions that utilize Intel SGX is due to the lack of asynchronous calls to and from the enclave.

As the number of transitions decreases, so does the execution time until 1 MiB, buffers higher than that increased execution time when executed under Intel SGX. We have no explanation for these results.

From the performed tests, we can also see that Graphene-SGX (Tsai et al., 2017) was the solution that provided the least impact on performance on this test, performing similarly to the application ported manually to SGX on buffers of size equal to 1 MiB or higher. Scone (Arnautov et al., 2016) performed slightly worse overall when compared to Graphene-SGX.

We were not able to run the application under SGX-LKL as it kept causing segmentation faults on *fread* and *fclose* systems calls.

5.2. *OpenSSL engine*

In this Section, we present the test case for our Intel SGX key store integrated with OpenSSL. For this we measure the overhead associated with our solution when utilizing RSA private keys.

5.2.1. *Methodology*

Our first engine test was based on OpenSSL's speed module, which benchmarked various algorithms within OpenSSL. We tested five different RSA key bit sizes, 512, 1024, 2048, 3072 and 4096. For each key size, 15,000 RSA SSL signatures were performed on 36 bytes of random data and repeated ten times so that we could take the average execution time and its associated error. The 36 bytes of random data was chosen as it was the value utilized in OpenSSL's benchmark. The test was executed on our solution and the default OpenSSL RSA implementation running outside an SGX enclave and within one using Graphene-SGX (Tsai et al., 2017), SGX-LKL (Priebe et al., 2019) and Scone (Arnautov et al., 2016).

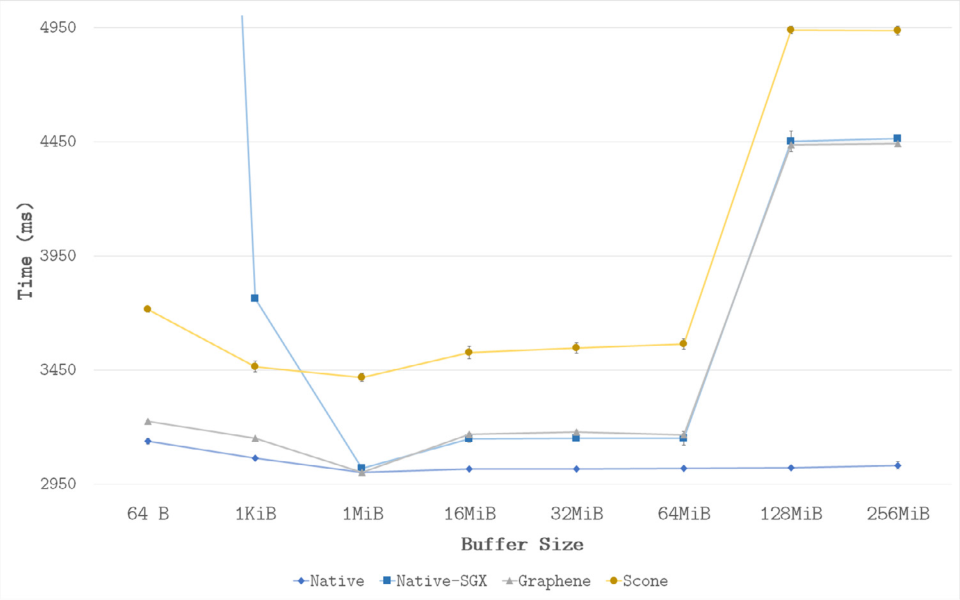

Fig. 7 – Average time (with sigma error bars) taken to hash 256 MiB of random data on disk with different buffer sizes.

Table 2 – RSA signatures per second on different solutions and various key sizes, in bits.

Solution \ Key size	512	1024	2048	3072	4096
Native	23286 ± 171	10646 ± 59	1685 ± 3	545 ± 1	252 ± 0
OpenSSLEngineSGX	13473 ± 247	7750 ± 63	1431 ± 41	523 ± 5	241 ± 1
Graphene-SGX	23156 ± 100	10520 ± 31	1684 ± 1	564 ± 0	252 ± 0
SGX-LKL	22088 ± 104	10504 ± 52	1615 ± 2	536 ± 0	240 ± 0
Scone	18785 ± 380	8795 ± 59	1491 ± 3	504 ± 1	226 ± 0

5.2.2. *Results*

In Table 2 we can see our solution described in Section 4.2 identified by ”OpenSSLEngineSGX” compared to the standard OpenSSL implementation running natively and under different environments with various RSA key sizes. When utilizing small key sizes, which results in a fast signature computation, our solution is significantly slower than the native solution, decreasing the number of signatures per second by 42.1%. On the more expensive key sizes, the decrease in performance is as low as 4.36%. SSL Labs shows that the most common key strength on Alexa’s list of the most popular sites in the world is 2048 bit (Qualys Inc, 2020), on which our solution performs 15.07% worse than the native solution without any protection. Verification of signatures is unaffected as it is an operation that uses the public key, which does not need to have the same level of protection as the private key, meaning it can run with its native implementation outside the enclave.

We can also see that our solution from the solution that utilizes SGX performs significantly worse up until RSA key sizes of 2048 bits. We believe this is due to an increase in enclave transitions and the lack of asynchronous calls. Solutions like Graphene-SGX, SGX-LKL, and Scone run the entire application inside the enclave, meaning there is no need to leave and enter the enclave on every operation. As the transitions to and from the enclave decrease, we can see that our solution’s relative performance approaches the native solution and beats both SGX-LKL and Scone.

We have also noted that, out of the solutions that aim to run unmodified applications inside an enclave, Scone seems to be the one that performed the worst in this test case, just like the one presented in Section 5.1.

5.3. *Apache web server - TLS*

This Section presents the test case for our implementations applied to the Apache web server when utilizing HTTPS. We test our implementations described in Section 4 along with some solutions presented in the state of art that aim to run unmodified applications within Intel SGX.

5.3.1. *Methodology*

We use a two Core 3.90 GHz Intel Pentium Gold G5600 with hyper-threading enabled with 8GB of dual-channel DDR4 memory as the client to benchmark the Apache webserver instances. The server is as indicated at the beginning of Section 5. Both machines are connected to the same local network, the client being connected through an SPF+ 10 Gbps card and the server utilizes a 1 Gbps Ethernet card. We chose to utilize a separate machine to run the benchmark tools so that the benchmark tool’s work would not interfere with the web server’s work, fighting for computational resources.

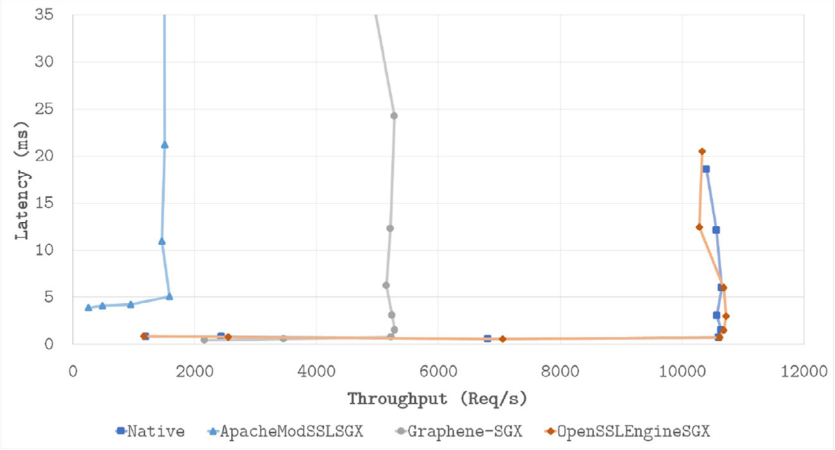

Fig. 8 – Throughput versus latency of Apache's web server workload.

To measure each solution's impact on the webserver, we utilize ApacheBench (Apache,) on the client machine to generate a workload on the server. On each test, the tool is executed nine times, performing 10000 requests and each time doubling the number of concurrency connections from the previous execution, except the last test, we used 196 concurrent requests instead of 256 as it started causing requests to be dropped. The tools measure the average throughput (requests per second) and the average latency of each request.

5.3.2. *Results*

In Fig. 8, we can see both of our solutions compared to the normal Apache webserver running normally and with Graphene-SGX (Tsai et al., 2017). We can see that our solution utilizing the custom OpenSSL Engine (OpenSSLEngineSGX in Fig. 8) shows performance similar to the original Apache web server, clearly outperforming Graphene-SGX while still guaranteeing integrity and confidentiality of the private key utilized to initialize the TLS connection. We can also see that the overhead shown in Table 2 is not enough to affect the Apache web server in our test, indicating that the bottleneck is somewhere else in Apache's web server.

Both the original Apache web server and our solution leveraging our key store through an OpenSSL Engine peak at approximately 10 700 requests per second. Graphene-SGX (Tsai et al., 2017) has its peak throughput cut in 51.4% at approximately 5 200 requests per second. Before the rise in latency, all these showed a latency smaller than a millisecond. The solution implemented in Section 4.1 is the solution (ApacheModSSLSGX in Fig. 8) that performed the worse. This can be explained by the significant increase in transitions to and from the enclave required during the normal operations of the webserver when requiring calls to the cryptography API. If we take the example of Fig. 5 in our solution, we have four enclave transitions, the call to *SSL_read* (to enclave), the call to *bio_filter_in_read* (from enclave), and the return of the previous two functions. Compared to Graphene-SGX, this input chain would cause two enclave transitions for the read system call. Additionally, we needed to restrict Apache to a single process due to forking issues regarding secure enclaves, causing further performance issues. All this leads to a minimum latency five times higher than the other solutions and a maximum throughput of 1500 requests per second.

We attempted to load the webserver in SGX-LKL, but we were not able to utilize it. We discovered that even though the server started listening on the specified ports and accepted the TCP connections, it did not reply with the contents of the page, and the connection just hanged.

Although Scone (Arnautov et al., 2016) benchmarked Apache's web server, the current solution (Scone,) does not provide a curated image for the Apache's web server. Compiling the webserver from source and executing it inside Scone results in the same behavior as SGX-LKL, the connection hanged.

Our test does not include TaLoS (Aublin et al., 2017) as we could not compile the webserver against it due to missing functions in the cryptography library when built-in hardware mode.

6. Threat model

In this section, we provide a threat analysis of the proposed system. We identify potential threats of each implementation and define how an attacker may attempt to exploit the system and the limitations we introduce to block that thread.

Key Negotiation

The impersonation attack on the HTTPS protocol is normally due to the unauthorized access to the private key on the server (Callegati et al., 2009) which can affect either current and previous communications. In order to mitigate this, we implemented OpenSSL inside SGX, giving an extra layer of security to the private and symmetric key management.

Physical access to the machine

If an attacker is able to compromise the server, in our first approach the attacker can modify the code of the web-server to read the data before and after it is encrypted and decrypted respectively. In our second implementation the previous attack would also viable, and they would also have access to the symmetric key established for the TLS connection as it is left exposed. This allows an attacker to be able to access the information of established connections but not the previous ones.

In this new scenario, the attacker will have clear limitations to access the private key, meaning that even if the system is compromised, the private key's security is never compromised. The attacker cannot, without enclave's knowledge, maliciously utilize the private key to authenticate themselves against other parties(e.g. establishing new connections to the clients or other micro-services).

DoS scenario

Denial-of-service attack limits the machine or network resource unavailable to its real users by temporarily disrupting services of a network component connected to the Internet. In this scenario, our implementation through an OpenSSL Engine focuses on mitigating the delay associated with manipulating the private key. As we can see in Section 5, this implementation has a negligible performance impact on the web server, meaning that your solutions would not be the bottleneck, by Intel SGX, in case of a DoS attack to the apache server.

7. Conclusion and future work

In this Section, we overview our work and results accomplished throughout this work. We start by making a brief overview of the research and development made throughout our work, commenting on the goals achieved and complexity of each project. We then talk about the results obtained, their effectiveness and viability. Lastly, we hint on possible future directions of this work, as an ongoing effort to minimize the impact on performance by secure enclaves.

7.1. Research and development

Despite our efforts, we struggled to find applications that utilized Intel SGX. Instead, we found many solutions that either automatically porting applications (Lind et al., 2017) or ran unmodified applications inside Intel SGX (Arnautov et al., 2016; Baumann et al., 2015; Priebe et al., 2019; Tsai et al., 2017). As seen by the results obtained in Chapter 5, some of these solutions introduce limitations that only allow certain types of applications to run inside SGX.

Given this state-of-affairs, we decided to port some applications to utilize Intel SGX through the SGX SDK and compare how these would perform relatively to their original counterparts running without any form of support from Intel SGX.

The Apache's web server can secure connections via TLS through a separate module called *mod_ssl*. Porting this module was harder than expected, taking a considerable portion of the time spent on this work. We encountered several issues, ranging from finding a compatible library with it and Intel SGX, to modifying the cryptography library to use it from untrusted code and through its OpenSSL compatibility layer.

Developing the Keystore for Intel SGX took significantly less effort than porting *mod_ssl*, with the harder task being modification of the OpenSSL Engine with proper documentation available.

Additionally, in Section 4.1, we also identified some issues with Intel's SGX cryptography library, TaLoS, and explained why it would not be suitable for our purposes.

7.2. Results

We successfully integrated the *mod_ssl* with WolfSSL and Intel SGX so that the termination of the TLS connection is made within a secure enclave to protect both the private key and the generated symmetric key during the handshake. However, this protection comes with a great performance penalty compared to both the original code of the module and Graphene-SGX, decreasing the performance as much as 90% and 70%, respectively.

Our second approach was aimed to guarantee secrecy of the private keys and involved implementing a Keystore in Intel SGX and integrating with an OpenSSL engine so that applications could use it transparently. Applying this solution to the Apache webserver resulted in no measurable overhead when compared to the unmodified version while performing significantly better than Graphene-SGX. This solution's caveat is that it does not protect the symmetric key agreed upon during the TLS handshake. Table 2 shows that there is, in fact, some overhead in our solution, but it was not significant enough to affect the web server.

We were limited to a single Intel SGX capable machine to test our solutions and the others. Additionally, we could not compile a Linux kernel with KVM supporting Intel SGX for guest virtual machines to analyze how Intel SGX would perform in a cloud-like scenario, limiting us to only being able to utilize docker.

Given our second approach versatility and performance, it is harder to recommend the first implementation, which has an inferior performance and may make it harder to port future updates made to the original *mod_ssl*. While the first solution does protect the TLS connection inside the secure enclave, an attacker still has access to the web server's code and memory, meaning the encrypted and decrypted data can be read on the read and write callbacks of the web server, which means the protection is not that much greater when compared to the implemented OpenSSL Engine.

7.3. Future work

Although we achieved our goals and utilized Intel SGX to increase key secrecy in real-world scenarios, our solution is not without limitations, pointed throughout this paper. In the future, we would like to address these limitations.

Asynchronous calls

All our solutions, for simplicity, implemented synchronous function calls to and from the enclave. We acknowledge that this is not ideal and brings a significant overhead to certain work scenarios. TaLoS (Aublin et al., 2017) has shown that the implementation of asynchronous has improved performance on their workload by as much as 117%. In Section 5.1, we can see that other solutions like Scone (Arnautov et al., 2016; Tsai et al., 2017), which implemented asynchronous function calls perform significantly better than our solution when a high amount of enclave transitions are made. To mitigate some of our solutions' performance penalty future work could implement asynchronous function calls.

Public-key cryptography support
Due to time constraints, the solution presented in Section 4.2 only implemented support for RSA public-key cryptography. Consequently, applications utilizing our engine are limited to RSA public-key cryptography. Future work of this solution should be able to support other public-key cryptography algorithms such as Elliptic-curve cryptography.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work of João S. Resende was supported by Fundação para a Ciência e Tecnologia (FCT), Portugal (PD/BD/128149/2016). This work has been supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe (cybersec4europe.eu) and project POCI-01-0247-FEDER-039598 (COP - Container Policing Hardening). Rolando Martins was partially funded by project POCI-01-0247-FEDER-041435 (SafeCities) and POCI-01-0247-FEDER-039598 (COP), financed by Fundo Europeu de Desenvolvimento Regional (FEDER), through COMPETE 2020 and Portugal 2020.

Appendix A. Read and write primitives within Intel SGX

Under normal circumstances the following function declarations would not be considered a security risk in a program as they perform normal operations necessary for the normal execution of the program. A problem arises however when these functions are exposed to untrusted code from the secure enclave, an attacker can pass an arbitrary pointer to the function, which will cause a read or write from memory, ultimately exposing the secure memory to untrusted code, which can be used to extract secrets from the enclave.

A1. *Read primitive*

```
int wolfSSL_get_fd(const WOLFSSL* ssl)
{
 int fd = -1;
 if (ssl) {
 fd = ssl->rfd;
 }
 return fd;
}
```

If we analyze at a lower level of what the processor does in this function we can see that it will add the offset of *rfd* in the structure *WOLFSSL* to the pointer provided by the variable *ssl*, read the contents of the resulting address, and return it. As C does give any guarantees that the passed pointer is actually of the type *WOLFSSL*, an attacker could pass an arbitrary pointer to this exposed function, allowing access to the otherwise inaccessible memory region of the enclave.

A2. *Write primitive*

```
int wolfSSL_CTX_set_TicketHint
 (WOLFSSL_CTX* ctx, int hint)
{
 if (ctx == NULL)
 return BAD_FUNC_ARG;
 ctx->ticketHint = hint;
 return WOLFSSL_SUCCESS;
}
```

Similarly to the read primitive, the method *wolfSSL_CTX_set_TicketHint* shows a write primitive to the secure memory if exposed to untrusted code. The attacker is able to write 4 bytes, the size of an integer to the address pointed by the variable *ctx* plus the offset of *ticketHint* in the *WOLFSSL_CTX* structure.

Appendix B. Developer notes

This appendix contains a few notes to help explain the logic of our implementations through this work.

B1. *Callback handling from trusted to unstrusted code*

```
//unstrusted code
int do_BIO_meth_read_cb(BIO bioId,
char *out, int inl, void* callback)
{
 int(*f)(BIO, char*, int) = callback;
 return f(bioId, out, inl);
}
//trusted code
int BIO_meth_set_read_callback_handler
 (WOLFSSL_BIO *bio, char *in, int inl)
{
 WOLFSSL_BIO_IDENTIFIER bioId =
 MAP_GET(WolfBioMapInverse, bio);
 WOLFSSL_BIO_METHOD* biom = bio->method;
 void** array =
 GetBioCallbackArray(biom);
 int retval = 0;
 do_BIO_meth_read_cb(&retval, bioId,
 in, inl,
 array[BIO_READ_CALLBACK_INDEX]);
 return retval;
}
int sgx_BIO_meth_set_read
(WOLFSSL_BIO_METHOD_IDENTIFIER biomId,
 void* callback)
{
 WOLFSSL_BIO_METHOD* biom =
 MAP_GET(WolfBioMethodMap, biomId);
 if(biom == NULL || callback == NULL)
 return WOLFSSL_FAILURE;

 void ** array =
 CreateBioCallbackArray(biom);
 array[BIO_WRITE_CALLBACK_INDEX] =
 callback;
 return wolfSSL_BIO_meth_set_read(biom,
 &BIO_meth_set_read_callback_handler);
}
```

The code snippet above shows how we handle the callbacks from trusted to untrusted code. As Intel SGX can not jump between trust and untrusted without first issuing a special instruction, steps must be taken to accommodate for this requirement. The SGX SDK partially abstracts this by creating proxy functions which replaces the return value that indicates if the call was successfulsgx (2021) and moving the real return value to the first parameter as a pointer. When the application wishes to register a callback the function *sgx_BIO_meth_set_read* is called, this function will store the callback in a callback array for the specified *BIO_METHOD* object and instead register a special stub that resided in trusted code with the cryptography library. When the stub function is called, we resolve the actual function to be called within untrusted code through the aforementioned array and forward the function pointer to a proxy function in untrusted code, ultimately calling the registered function.

B2. *OpenSSL engine configuration*

```
[openssl_init]
engines=engine_section
[engine_section]
sgxkeystore = sgxkeystore_section
[sgxkeystore_section]
engine_id = sgxkeystore
dynamic_path = /usr/lib/x86_64-linux-gnu
/engines-1.1/sgxkeystore.so
init = 0
```

For the solution in Section 4.2 to be usable we must tell OpenSSL about our engine, this can be done by modifying its configuration, in our case in the file */etc/ssl/openssl.cnf*. In this file we specify which engines to be aware of on initialization, its id, the path of the binary and if the engine should be automatically initialized or if the application should do it.

B3. *Utilizing an OpenSSL engine from command line*

```
$ openssl rsautl -engine sgxkeystore
 -keyform engine -inkey
 sgxkeystore:testkey.pem.sealed
 -decrypt -in key.b in.enc -out key.bin2
$ openssl dgst -engine sgxkeystore
 -keyform engine -sign
 sgxkeystore:testkey.pem.sealed
 -out signature.bin -sha256 foo.txt
```

After configuring the engine, it can even be used via command line to perform cryptographic operations. In order to do that the engine must be specified as well as the key format, via the *-engine* and *-keyform* options.

B4. *Patch to support another engine on apache web server*

```
--- ssl_util.c
+++ ssl_util.c.sgxkeystore
@@ -477,7 +477,7 @@
 {
 #if defined(HAVE_OPENSSL_ENGINE_H) \
 && defined(HAVE_ENGINE_INIT)
 /* ### Can handle any other
 special ENGINE key names here? */
- return strncmp(name, &#x27;&#x27;pkcs11:&#x27;&#x27;;, 7) == 0;
+ return strncmp(name, &#x27;&#x27;pkcs11:&#x27;&#x27;;, 7) == 0
+ || strncmp(name, &#x27;&#x27;sgxkeystore:&#x27;&#x27;;, 12)==0;
 #else
 return 0;
 #endif
```

Since Apache's web server already contains support for pkcs#11 devices and because it uses the prefix before the colon as the engine id, adding support for our engine required only a simple patch to include our engine id as a known engine.

REFERENCES

Apache,. Apache http benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

Arnautov S, Trach B, Gregor F, Knauth T, Martin A, Priebe C, Lind J, Muthukumaran D, O'Keeffe D, Stillwell ML, et al. {SCONE}: Secure Linux containers with Intel {SGX}. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16); 2016. p. 689–703.

Aublin P-L, Kelbert F, Okeeffe D, Muthukumaran D, Priebe C, Lind J, Krahn R, Fetzer C, Eyers D, Pietzuch P. Talos: Secure and transparent TLS termination inside SGX enclaves. Imperial College London, Tech. Rep 2017;5:2017.

Bauman E, Wang H, Zhang M, Lin Z. Sgxelide: enabling enclave code secrecy via self-modification. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization; 2018. p. 75–86.

Baumann A, Peinado M, Hunt G. Shielding applications from an untrusted cloud with haven. ACM Trans. Comput. Syst. (TOCS) 2015;33(3):8.

Brasser F, Müller U, Dmitrienko A, Kostiainen K, Capkun S, Sadeghi A-R. In: 11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17). Software grand exposure:{SGX} cache attacks are practical; 2017.

Brandão A., 2020a. Apache mod_ssl with sgx. GitHub Repositroy, https://github.com/andreluis034/module_ssl_sgx/.

Brandão A., 2020b. Sgx keystore openssl engine. GitHub Repository, https://github.com/andreluis034/sgx-keystore.openssl-engine.

Callegati F, Cerroni W, Ramilli M. Man-in-the-middle attack to the https protocol. IEEE Secur. Privacy 2009;7(1):78–81.

Cloosters T, Rodler M, Davi L. In: 29th USENIX Security Symposium (USENIX Security '20). TeeRex: discovery and exploitation of memory corruption vulnerabilities in SGX enclaves; 2020.

Costan V, Devadas S. Intel SGX explained.. IACR Cryptol. ePrint Arch. 2016;2016(086):1–118.

Domas C. The Memory Sinkhole. BlackHat USA; 2015.

Donenfeld JA. In: NDSS. Wireguard: next generation kernel network tunnel.; 2017.

Fontaine C, Galand F. A survey of homomorphic encryption for nonspecialists. EURASIP J. Inf. Secur. 2007;2007:1–10.

OpenSSL, 2017. OpenSSL 1.1.0 changes. OpenSSL wiki https://wiki.openssl.org/index.php/OpenSSL_1.1.0_Changes. Accessed 2020-05-26

Google, 2020. Android keystore system | android developers. https://developer.android.com/training/articles/keystore. Acessed on 2021-03-27.

GlobalPlatform, 2017. TEE System Architecture v1.2. https://globalplatform.org/specs-library/?filter-committee=tee, Accessed: 2020-01-12

Google, 2015. Android Open Source Project. https://android.googlesource.com/platform/system/security/.

Guide, P., 2011. Intel® 64 and IA-32 architectures software developers manual. Volume 1 1.

Hashicorp,. Vault by hashicorp. https://www.vaultproject.io/ Acessed on 2021-03-27.

Scone,. - a secure container environment.https://scontain.com/index.html

Intel 2021software guard extensions (Intel SGX) SDKfor Linux* os. https://download.01.org/intel-sgx/sgx-linux/2.13/docs/Intel_SGX_Developer_Reference_Linux_2.13_Open_Source.pdf, Acessed on 2021-03-21.

Intel. In: Volume 3B: System Programming Guide, Part 3B. Intel® 64 and IA-32 architectures software developers manual; 2011.

Intel. Intel software guard extensions SSL library. arXiv preprint arXiv:1908.11143 2017.

Krawiecka K, Kurnikov A, Paverd A, Mannan M, Asokan N. Safekeeper: protecting web passwords using trusted execution environments. In: Proceedings of the 2018 World Wide Web Conference; 2018. p. 349–58.

Kurnikov A, Paverd A, Mannan M, Asokan N. Keys in the clouds: Auditable multi-device access to cryptographic credentials. CoRR 2018;abs/1804.08569.

Lee J, Jang J, Jang Y, Kwak N, Choi Y, Choi C, Kim T, Peinado M, Kang BB. Hacking in darkness: return-oriented programming against secure enclaves. In: 26th {USENIX} Security Symposium ({USENIX} Security 17); 2017. p. 523–39.

Lee J, Jang J, Jang Y, Kwak N, Choi Y, Choi C, Kim T, Peinado M, Kang BB. Hacking in darkness: return-oriented programming against secure enclaves. In: 26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association; 2017. p. 523–39.

Liang H, Li M, Chen Y, Jiang L, Xie Z, Yang T. Establishing trusted i/o paths for SGX client systems with aurora. IEEE Trans. Inf. ForensicsSecur. 2019;15:1589–600.

Qualys Inc., 2020. SSL Pulse. Online https://www.ssllabs.com/ssl-pulse/. Acessed 2020-07-05

OpenEnclave, 2020. GitHub repository. https://github.com/openenclave/openenclave

Weisse, O., Bulck, J.V., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M. Strackx, R. Wenisch, T.F Yarom, Y. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-order Execution, Technical report See also USENIX Security paper Foreshadow[19] (2018).

Lind J, Priebe C, Muthukumaran D, O'Keeffe D, Aublin P-L, Kelbert F, Reiher T, Goltzsche D, Eyers D, Kapitza R, et al. Glamdring: automatic application partitioning for Intel {SGX}. In: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17); 2017. p. 285–98.

Minkin M, Moghimi D, Lipp M, Schwarz M, Van Bulck J, Genkin D, Gruss D, Piessens F, Sunar B, Yarom Y. Fallout: reading kernel writes from user space. arXiv preprint arXiv:1905.12701 2019.

OpenDNSSEC,. Softhsm. https://www.opendnssec.org/softhsm/ Acessed 2020-04-10.

OpenSC, 2020. libp11. https://github.com/OpenSC/libp11.

musl. Musl C Library. Online; 2019 https://musl.libc.org/about.html.

Porter DE, Boyd-Wickizer S, Howell J, Olinsky R, Hunt GC. Rethinking the library os from the top down, 39. ACM; 2011. p. 291–304.

Priebe C, Muthukumaran D, Lind J, Zhu H, Cui S, Sartakov VA, Pietzuch P. SGX-LKL: securing the host os interface for trusted execution. arXiv preprint arXiv:1908.11143 2019.

Oscarlab, 2019. Graphene. GitHub repository https://github.com/oscarlab/graphene

Richter L, Götzfried J, Müller T. Isolating operating system components with Intel SGX. In: Proceedings of the 1st Workshop on System Software for Trusted Execution; 2016. p. 1–6.

Sgx-lkl 2019. GitHub repository.

Sabt M, Achemlal M, Bouabdallah A. Trusted execution environment: what it is, and what it is not, 1. IEEE; 2015. p. 57–64.

van Schaik, S., Kwong, A., Genkin, D., Yarom, Y., 2020a. SGAxe: how SGX fails in practice. https://sgaxeattack.com/.

van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y., 2020b. Cacheout: leaking data on Intel cpus via cache evictions. 2006.13353.

Schuster F, Costa M, Fournet C, Gkantsidis C, Peinado M, Mainar-Ruiz G, Russinovich M. Vc3: trustworthy data analytics in the cloud using SGX. In: 2015 IEEE Symposium on Security and Privacy. IEEE; 2015. p. 38–54.

Schwarz M, Lipp M, Moghimi D, Van Bulck J, Stecklina J, Prescher T, Gruss D. In: CCS. ZombieLoad: cross-privilege-boundary data sampling; 2019.

Tian D, Choi JI, Hernandez G, Traynor P, Butler KR. A practical Intel SGX setting for Linux containers in the cloud. In: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy; 2019. p. 255–66.

Tsai C-C, Arora KS, Bandi N, Jain B, Jannen W, John J, Kalodner HA, Kulkarni V, Oliveira D, Porter DE. Cooperation and security isolation of library OSes for multi-process applications. In: Proceedings of the Ninth European Conference on Computer Systems. ACM; 2014. p. 9.

Tsai C-C, Porter DE, Vij M. Graphene-sgx: a practical library {OS} for unmodified applications on {SGX}. In: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17); 2017. p. 645–58.

Van Bulck J, Minkin M, Weisse O, Genkin D, Kasikci B, Piessens F, Silberstein M, Wenisch TF, Yarom Y, Strackx R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. Proceedings of the 27th USENIX Security Symposium. USENIX Association, 2018. See also technical report Foreshadow-NG Weisse et al. (2018)

Weichbrodt N, Kurmus A, Pietzuch P, Kapitza R. Asyncshock: exploiting synchronisation bugs in Intel SGX enclaves. In: European Symposium on Research in Computer Security. Springer; 2016. p. 440–57.

WolfSSL .Wolfssl. https://www.wolfssl.com/, Online; Acessed on 07-05-2020.

André Brandão obtained his M.Sc in Network and Information Systems Engineering at the Faculty of Science of the Universide of Porto (FCUP). In virtue of his master's thesis, "Systems Hardening through the use of Secure Enclaves", he joined C3P under the supervision of Rolando Martins and João S. Resende. During the first of year of his master's he worked with INESC TEC to study on the application of cheat detection hardening using secure enclaves.

João S. Resende is an invited assistant at the Department of Computer Science at the University of Porto. He is the project manager and researcher on the project CyberSec4europe. CybersSec4europe is one of the four pilots focusing on the creation of a Network of Cybersecurity Competence and a new European Cybersecurity Competence Centre in Romania to improve cybersecurity capacity. Currently, he is finishing his Ph.D. in Computer Science in a collaborative effort between three Portuguese Universities Minho, Aveiro, and Porto under the principal supervision of Luís Antunes and Rolando Martins, where he was exposed to the difficulties underlying today's critical systems of Portugal.

Rolando Martins is an Assistant Professor at the Computer Science Department of the Faculty of Science of the University of Porto (FCUP). He is currently a senior research within the Cybersecurity and Privacy Centre, currently focusing on secure distributed systems and privacy supporting infrastructure. He is a former member of the the Intel Science and Technology Center (ISTC), where he was involved in both Cloud Computing and Embedded Computing centers, and Parallel Data Lab (PDL) at Carnegie Mellon University (CMU). At the same time, he was also a computer research scientist at YinZcam, a spinoff from CMU that provided mobile applications for the NBA, NHL, NFL and MLS, where he was involved on cloud computing, big data, content management systems, security and video streaming. He was a former consultant for Porto City Hall in their GDPR adoption, for nearly two years, having had responsibilities in overseeing their security implementation and processes in order to achieve compliance. He is currently the lead consultant for INESC-TEC in Efacec's transition to cloud computing, adoption of new cybersecurity policies and artificial intelligence. He is currently on INESC-TEC's technical advisory board for networking and infrastructure. Lastly, he is also consultant for security, cloud and mobile technology for Adyta, a leading security startup in secure messaging and infrastructure. He is currently the PI for Container Oriented Policing (COP) research project, a P2020 copromotion project with Talkdesk, that seeks to enhance current security and monitoring for container technology for commercial adoption by Talkdesk. He was also involved in TRONE, a CMU-Portugal project with CMU, Faculty of Sciences of University of Lisbon and Portugal Telecom towards securing datacenters using byzantine fault-tolerance to hardness the information harvested by PT's Security information and event management (SIEM).