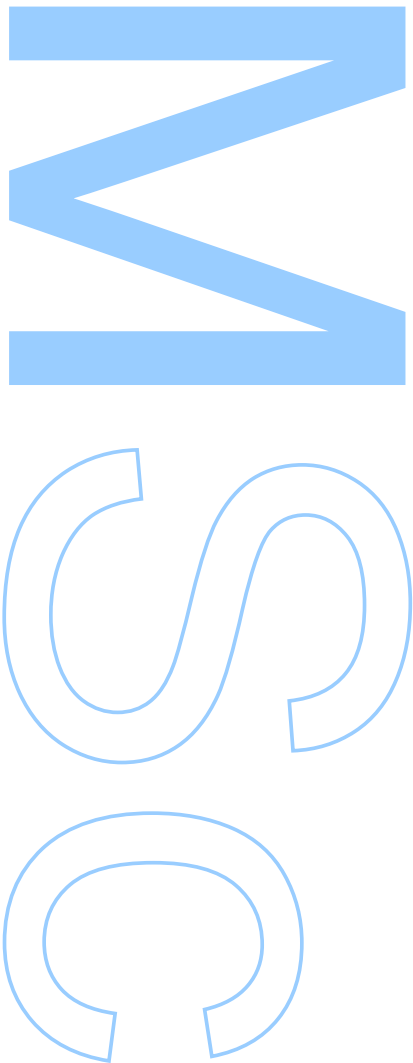
Systems Hardening through the use of Secure Enclaves

André Luís Godinho de Sousa Brandão
Mestrado Integrado em Engenharia Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2020

Orientador
Rolando da Silva Martins
Professor Auxiliar
Faculdade de Ciências da Universidade do Porto

Coorientador
João Miguel Maia Soares de Resende
Professor Assistente Convidado
Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

UNIVERSIDADE DO PORTO

MASTERS THESIS

Systems Hardening through the use of Secure Enclaves

Author:
André Luís Godinho de Sousa Brandão

Supervisor:
Rolando da Silva Martins

Co-supervisor:
João Miguel Maia Soares de Resende

A thesis submitted in fulfilment of the requirements
for the degree of MSc. Network and Information Systems Engineering

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

December 15, 2020

Acknowledgements

I want to thank my thesis supervisor, Professor Rolando Martins for the opportunity to work in this thesis topic that I wanted to pursue.

I also want to thank my thesis co-supervisor, João Resende for all the recommendations made and helping me through even the most of ridiculous questions that I had, during the research and development of this thesis.

This work has been supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe (cybersec4europe.eu).

I would also like to thank not only my friends that I made throughout my bachelor's and master's degree, but also those who I still in keep touch from high school through TeamSpeak and discord, wasting countless hours playing online video games.

Finally, I would like to thank my parents and my grandparents, for supporting me throughout my life in every possible way and always believing in me.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

MSc. Network and Information Systems Engineering

Systems Hardening through the use of Secure Enclaves

by André Luís Godinho de Sousa Brandão

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 or personal data when these are utilized in components controlled by a third-party vendor.

Secure enclaves can solve data theft problems by creating a secure environment where code can be executed securely, guaranteeing that no unwanted parties read the data inside the environment. During our research, we found a lack of applications that utilize secure enclaves in real-world scenarios.

In this thesis we explore the application of secure enclaves in different application domains, namely the hardening of web servers through the modification of the Apache web server to further protect its private key from the operating system and hypervisor, and additionally, we explore application introspection using secure enclaves to protect video games from cheaters, and test it against a cheat used in the real world in Counter-Strike: Global Offensive.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos

Fortificação de sistemas através do uso de enclavos seguros

por André Luís Godinho de Sousa Brandão

Com o aumento da utilização de serviços na Cloud, é perdido o controlo não só do hardware, mas também do código mais privilegiado, que são responsáveis por hospedar e correr os seus dados e software. Isto significa que as empresas estão a aceitar o risco de poderem perder dados confidências ou pessoas para terceiros.

Enclaves seguros permitem solucionar o problema do roubo de dados, através da criação de um ambiente segure onde código pode ser executado, garantindo que terceiros não tenham acesso aos dados de esse mesmo ambiente. Durante a nossa pesquisa reparamos na falta de aplicações que utilizam este tipo de ambientes seguros no mundo real.

Nesta tese exploramos a aplicação de enclavos seguros em diversos domínios, mais concretamente a fortificação de servidores web através da modificação do servidor web da Apache para proteger as chaves privadas do sistema operativo e do hypervisor, adicionalmente, explorámos introspeção aplicacional com enclaves seguros para proteger vídeo jogos de cheaters e testámo-la contra um cheat utilizado no mundo real no Counter Strike: Global Offensive.

Contents

List of Figures

List of Tables

Glossary

ABI	Application Binary Interface
API	Application Programming Interface
EPC	Enclave Page Cache
LKL	Linux Kernel Library
OS	Operating System
PAL	Platform Abstraction Layer
PRM	Processor Reserved Memory
SGX	Software Guard eXtensions
SMM	System Management Mode
SSL	Secure Sockets Layer
SoC	System on Chip
TLS	Transport Layer Security
TPM	Trusted Platform Module
VMM	Virtual Machine Monitor
VPN	Virtual Private Network
CS:GO	Counter Strike: Global Offensive
DRM	Digital Rights Management
EPT	Extended Page Table
GOT	Global Offsets Table
GnuPG	GNU Privacy Guard
HSM	Hardware Security Module

HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
IDS	Intrustion Detection System
KPP	Kernel Patch Protection
MSBDS	Micro architectural Store Buffer Data Sampling
PLT	Procedure Linkage Table
ROP	Return-Oriented Programming
SDK	Software Development Kit
SEV	Secure Encrypted Virtualization
SE	Secure Element
SMT	Simultaneous Multi-Threading
TCB	Trusted Compute Base
TEE	Trusted Execution Environment
TSX	Transactional Synchronization Extensions
TXT	Trusted eXecution Technology
URI	Uniform Resource Identifier
WTF	Write Transient Forwarding

Chapter 1

Introduction

Some of contents of this thesis have been submitted to the *Computers & Security* journal, which is still under review as of the publication of this thesis. Additionally, the argumentation given in section 4.3 and its corresponding related work have been published [1] on the TrustBus 2020 conference.

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. The gravity of omitting these increases when considering rented hardware, as the user cannot be sure what code is executing in more privileged modes. A malicious hypervisor could modify and read any physical memory present in the machine; a malicious kernel could do the same to any memory allocated to user processes. This can arise issues such as non-encrypted data accessed in any storage type.

The issue above can be partially solved with homomorphic encryption, a field of cryptography 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 symmetric-key homomorphic encryption, and those that exist have, for the most part, been broken[2]. There are secure asymmetric-key homomorphic encryption schemes implemented but suffer from performance penalty

when compared to their symmetric counterpart.

An alternative solution to the problem is the use of trusted technologies. Some examples 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 in a way or another secure computation. The only limitation of these technologies is the requirement of 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 generic processors 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, allowing applications to consider a "wider" threat model, as it is, in theory, protected against privileged code.

1.1 Motivation

General-purpose computing devices nowadays run all sorts of software, each one of them with their vulnerabilities. This means that compromising one of them may lead to access to the device, potentially compromising the remaining software in it.

Intel SGX gives the developers stronger guarantees that the code executed was not tampered, while providing confidentially of the data and code inside it. It also 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 with many companies moving their infrastructure to administratively 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 [3] or running unmodified applications inside an enclave[4–7]. The issue with these approaches is that they do not consider each application's specifics and can result in a significant performance loss or even not work at all.

As Intel SGX prevents the tampering of the code executed in an untrusted host, we also explore on how we can implement tampering detection techniques on applications that

cannot reliably be ported to Intel SGX, such as video games, by working very similarly to existing client-side anti-cheat monitoring techniques that are often exploited by misplaced trust [8].

1.2 Objectives

The main goal of this thesis is to apply Intel SGX in two real-world scenario applications in order to guarantee integrity, confidentiality of confidential data and detection of application tampering. In summary, the objectives of this thesis can be resumed to:

- Identify existing server applications used in the real world and solutions that can leverage Intel SGX;
- Application of Intel SGX in applications used in the real world;
- Analyze monitoring possibilities and tamper prevention on sensitive applications (e.g. video games);
- Usage of a complementary kernel module to monitor file access, achieve debug prevention and unauthorized reads/writes on a process;
- Creation and deployment of prototypes, utilizing Intel SGX, in a real-world scenarios
- Definition of a threat model with security analysis for each scenario;

1.3 Outline

Firstly this chapter presented the motivation for this work and the objectives to be achieved in this thesis.

In chapter 2, we introduce some concepts necessary throughout this thesis. 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 chapter also explained what a trusted execution environment is and how it can break the typical security ring in modern processors.

In chapter 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. Additionally, we explore some existing commercial game anti-cheat solutions, and one solution leveraging Intel SGX.

Chapter 4 presents our work on the Apache's web server, the creation of an OpenSSL Engine, and the application of anti-cheat monitoring techniques to Intel SGX. Also, with each implementation, we perform a security analysis.

In chapter 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.

Finally, chapter 6, concludes this thesis by overviewing the results obtained in the previous chapter. Besides, we discuss some limitations of the current implementations and possible future iterations of this work.

Chapter 2

Background

In this chapter, 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. Finally, we introduce some basic concepts of Intel SGX that will be necessary throughout this thesis.

2.1 Security rings

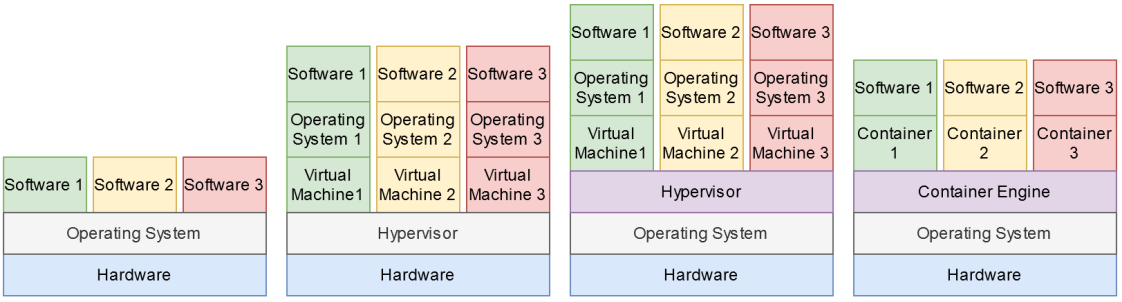
Modern processors typically have several execution modes that provide hierarchical layers of privilege, also known as the security rings [9]. The lower the ring in which the CPU mode operates, the higher the privilege is. Even though x86 gives various security rings to work with, general-purpose operating 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[10]. 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 from higher privileged code.

2.2 Virtualization

Virtualization is the concept of creating a virtual version of something. This is usually achieved by an abstraction layer that allows the creation of virtual ”hardware”. In this section, we will approach solutions that virtualize the hardware the operating system runs off.



(a) Typical Operating System (b) Hypervisor type 1 (c) Hypervisor type 2 (d) Virtualization via containers

FIGURE 2.1: Common virtualization types

Virtualization via Virtual Machines

To be able to create virtual machines, the bare machine must utilize a Virtual Machine Monitor (VMM), also commonly known as a hypervisor. The VMM has full control of the machine resources and aims to create an efficient, isolated virtual machine (b and c in figure 2.1) duplicate of the real machine[11], and this is accomplished by creating a Basic Machine Interface[12] for each virtual machine. Buzen and Galglirdi [12] define the primary machine interface as the set of all software visible objects and instructions supported by the system’s hardware and firmware.

Nowadays an hypervisor can be either of type one or type two. Type one are also known as bare metal hypervisors (b in figure 2.1), meaning there is no other operating system running parallel or beneath the hypervisor, allowing direct access to the hardware without any overhead. As the hypervisor itself is the ”Operating System” installed on the physical machine these type of hypervisor generally either have no interface for the user or a very limited one, requiring another machine to configure it and to deploy virtual machines.

Type two hypervisors on the other hand run above a ”normal” operating system such as Windows or Linux and exist mainly for convenience for the end-user to run a virtual

machine without requiring a dedicated machine for it. Memory allocation and I/O is handled by the operating system, providing an increased overhead when compared to type 1 hypervisors. Type 2 hypervisors may be less flexible on the resources and capabilities they provide as it is dependent on the underlying operating system.

The only way for a malicious or compromised virtual machines to affect others is to attack first the hypervisor or the processor to break the isolation barrier provided, the latter can often be attacked by leveraging, for example, side-channel vulnerabilities to extract information from other virtual machines or even the hypervisor.

Virtualization via virtual machines has become quite popular as it allows for the same software to run on a panoply of different hardware, without requiring changes to the software. They are allowing at the same time, if the processor supports, to isolate virtual machines at the hardware level completely.

Virtualization via Containers

Recently there has been an uprise in the interest of lightweight virtualization through containerization (d in figure 2.1). The kernel is responsible for isolating the different containers and protecting itself from it.

The containers share the kernel with the host, and this restricts the operating system that can run under the virtualization to one that shares the same kernel. It also means a bug in the kernel may compromise all the containers associated with it and the host operating system.

The isolation, on linux, is performed via *cgroups* and *namespaces*. Cgroups allow for the kernel to limit and measure the resources utilized by a group of processes, while namespaces allow the kernel limit the visibility the processes have of the rest of the system. Because only one kernel is used, the performance and size overhead are significantly reduced compared to a virtual machine.

Virtualization via library OS

A Library OS is a library loaded into the user space of a process providing as many kernel-like features as possible from userspace, only transitioning to kernel space when it is required, i.e. interfacing with the hardware.

The great advantage of this approach compared to containers is that it dramatically reduces the number of system-calls that kernel must defend itself from the application. It

may also be possible to create a host OS-dependent, abstraction layer between the library OS and the actual kernel, allowing the same executable to run in completely different environments (e.g. Linux and Windows) or even micro kernels that provide minimal functionality.

As the virtualization provided by a library OS is made purely through user space, the security boundaries it provides are essentially none, as an attacker can read the whole memory space in the process and nothing stops them from issuing syscall instructions to interact directly with the kernel without going through the library OS.

2.3 Trusted Execution Environment

Trusted Execution Environment (TEE) is often referred to, generally, as a secure, integrity-protected programmable environment with memory and sometimes storage capabilities [13]. Global Platform defines a ”TEE system architecture” [14] 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

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

2.3.1 Intel SGX

In 2015, Intel introduced, along with the Skylake microarchitecture, Software Guard Extension (SGX), which is 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[15].

Memory structure

The PRM holds Enclave Page Cache (EPC) sets, each with 4KB 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 [16], 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[15]. 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 90MB.

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[17, 18] 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[19][20]. 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, Jaehyuk Lee et al [21] 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 portions of the previous L1 cache 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 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.

Ferdinand Brasser et al [22] 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, uninterrupted execution of the enclave 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 necessity of the kernel never interrupting the core on which both the victim's code and attacker's code run.

Foreshadow[23] 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 the identity of

its signer. 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 Fallout[24] 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 so that it can acquire exclusive access to it. To prevent stalling the store instruction, the WTF optimization store the address and value in a buffer and continue 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 value from the buffer so that stales 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 Micro-architectural Store Buffer Data Sampling (MSBDS).

Zombieload[25] 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[26] demonstrates that Intel's fix on 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[27], 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.

2.3.2 OpenEnclave

OpenEnclave[28] is an hardware agnostic Software Development Kit (SDK) for trusted execution environments. Currently it only supports Intel SGX and Arm TrustZone. This allows for any developer to support a multitude of enclave solutions without worrying about the specifics of each system. It automatically partitions applications in two components, one to be executed inside the enclave and other outside the enclave for operations that are not permitted inside the enclave (e.g. system calls).

Chapter 3

Related work

In this chapter 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 the limited instruction set of a secure enclave. Secondly, we analyze some applications that have been created with Intel SGX in mind, which result in a much smaller code sizes than the previous solutions. Finally, we discuss secure introspection and how it may be applicable to Intel SGX.

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[3] or running unmodified applications inside Intel SGX[4–7]. In this section, we approach some solutions that aims to run unmodified applications inside the enclave and identify the key differences in each solution.

3.1.1 Scone

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. In this section, we will look at a few solutions that aim to run unmodified applications on Intel SGX.

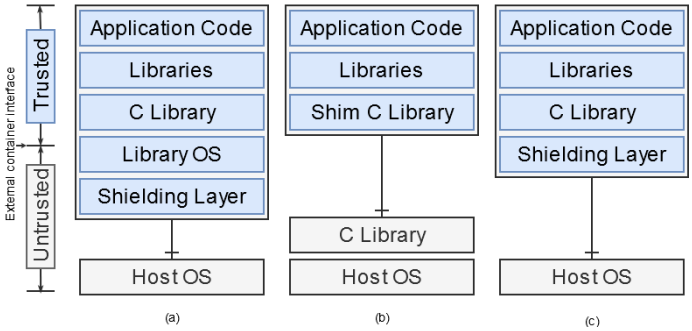


FIGURE 3.1: Secure container designs[4]

While state of the art solutions achieve the same objective, of running unmodified applications inside Intel SGX, this can be achieved in various ways. Figure 3.1 shows three possible ways that achieve similar results.

Figure 3.1a shows a design that places a Library OS and a shielding mechanism inside the enclave. The library OS allows for 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. Figure 3.1b 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 actual 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 overall performance of the application.

Finally, Figure 3.1c 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.

3.1.2 Haven

Haven[5] was the first solution to implement this kind of paradigm and follows the implementation in 3.1a. It allowed to run unmodified applications shielded from the operating system. To achieve this Haven builds on top of Drawbridge[29] a system supporting

sandboxing of Windows applications leveraging two mechanisms, microprocessors and a library OS.

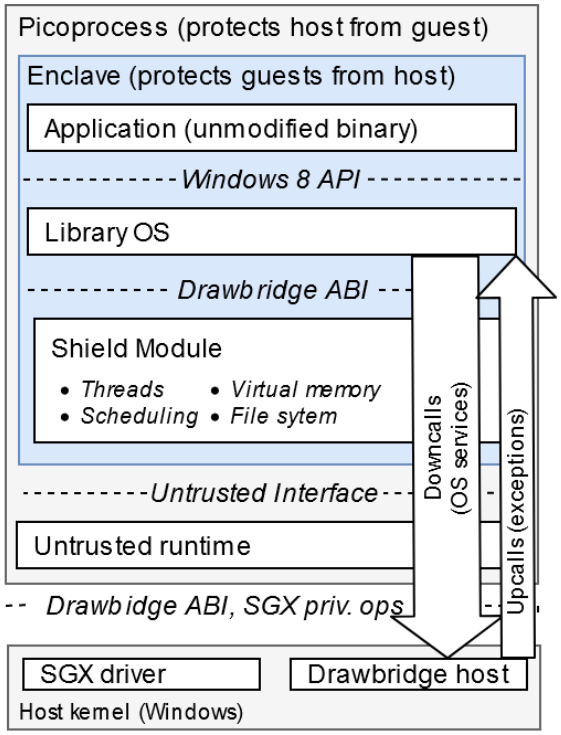


FIGURE 3.2: Haven components and interfaces[5]

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.

3.1.3 Scone

Scone[4] is a Linux solution similar to Haven, with the exception that it does not require a library OS to be loaded to the TCB (Figure 3.1c). To support applications inside the enclave, it runs inside the TCB, a modified musl C library[30]. 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[31]. 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[4] an optional kernel module is mentioned to improve system calls performance, but the website[31] for Scone does not mention this feature.

3.1.4 SGX-LKL

SGX-LKL[7] 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[32] 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 is not generated 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.

3.1.5 Graphene

Graphane[33] 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[6] treats 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 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[34] is 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-sided 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 replaced 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 that has 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.

To securely transmit the user's password, the browser add-on needs to correctly identify that it is communicating with the SafeKeeper password protection service. The identification of the service is done 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 creditianls will be sent.

3.2.2 Intel SGX key store

Keys in the Clouds is a solution presented by Arseny Kurnikov et al [35] 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 securely store and utilize the key, 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 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 difficuly, 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.

Dave Tian et al. [36] 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[5] that requiring moving a big code base to the enclave. Unlike traditional TPMs, it does not allow the use of the TPM before the operating system is initialized, meaning it cannot be used for example for secure boot. This service allows for multiple a

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 that it is communicating with a genuine Intel SGX enclave.

3.2.4 SGX-Kernel

Although Intel SGX is limited to running code in user mode, Lars Richter et al. [37] 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 communicated 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.

3.3 Secure introspecting

One way to protect the deployed systems is via introspection of the system itself. As the hypervisor is generally the most privileged mode that generic software, introspection systems are typically deployed at this level [38], with solutions like Bitdefender Hypervisor introspection[39] and Microsoft's Hypervisor-Protected Code Integrity.

One other common application of secure introspection is in video games, through the use of "Anti-Cheats", to protect the code integrity and guarantee fair play in multiplayer matches, these solutions are typically deployed in user or kernel space. They are often an extension to the base game, and depending on what it wants to monitor, it can run on the client, server, or even in both places. Since anti-cheats on the server-side are limited to analyzing the client's input, it is common to deploy an anti-cheat at both ends. Alternatively, it can just be implemented on the client-side as it permits to monitor the client's behavior in more detail. The disadvantage of this option is that it is exploited by misplaced trust [8] in the anti-cheat. The attacker often has full access to the device, to tamper with the running software, including but not limited to, the game code and the anti-cheat itself. This full access can be used to bypass the implemented security checks in the anti-cheat, giving the illusion that nothing has been tampered.

Sometime after the release of Intel SGX, Erick Bauman et al. [40] introduced a solution to utilize SGX to protect computer games and, at the same use it as a digital rights management (DRM) system. The proposed protection model to prevent players from cheating requires some of the game's code and data to be moved to the enclave created by Intel SGX. As the Intel SGX Enclave Page Cache (EPC is limited at maximum to 128 MB of memory [16] and does not permit the usage of system calls without first exiting the enclave. The game developer must first perform a careful analysis of the game's code not to exceed the EPC size to degrade performance beyond an unusable point. While this might

be easy for games with a small code base, this task gets harder with the size of code that must be protected. As the system grows, the likelihood of game code that might leak data to and from enclaves will increase. While it might not seem an issue at first because tools such as Cheat Engine are still unable to read game data inside the enclave, it is not enough to defend from an application that might load additional code in the application [41, 42].

To our knowledge, there does not exist a commercial anti-cheat solution that takes advantage of Intel SGX to implement cheat detection techniques. Current anti-cheats solutions work in diverse ways from just server-side monitoring to kernel space monitoring in the attacker's device. However, almost all typically rely partially on security by obscurity, probably with fear that attackers may more easily find exploits to circumvent these.

Battleye

Battleye [43] works by running code on the client's device and the server. With the main component being on the client's side, and using the servers only for reporting the cheater. It takes advantage of the client's device by running code in both user and kernel space to scan not only the running process but also other processes and impede code injecting on the game's process.

Easy Anti-Cheat

Kamu.GG was released initially and later acquired by Epic Games [44], Easy Anti-Cheat [45], which works similarly to Battleye and achieves the same results in the same or similar fashion. Working in both kernel and userspace gives the anti-cheat access to the game process and others to monitor for unwanted activity. This anti-cheat will not load if specific Windows features that aid debugging of kernel code or loading non-signed windows drivers are enabled. Such options include disabling Driver Signature Enforcement, Kernel Debugging, Windows Safe mode or disabling Kernel Patch Protection (KPP), Microsoft's solution to prevent patching the Windows NT Kernel.

Fairfight

Fairfight [46] by GameBlocks, LLC, is a non-intrusive, server-sided anti-cheat, that works much like a behavior-based Intrusion Detection System (IDS). The anti-cheat interprets the player's game events and uses them to compare against the rest of the players. The

system may flag those who perform exceptionally well or in a weird way (e.g., pointing to other players through opaque objects) for manual review by the developers. The advantage of this type of anti-cheat is that it allows catching unknown or new types of cheats, but it might also not catch every cheater, especially the ones playing within what is considered 'normal' behavior.

Chapter 4

Design and Implementation

In this chapter we describe our implementations of tamper prevention of sensible data, particularly encryption keys and a mechanism created to detect tampering of applications that cannot reliably be executed within Intel SGX.

First, we introduce a modification to the Apache's web server where the TLS termination is moved to a secure enclave. Secondly, in section 4.2 we present an encryption key storage implemented in Intel SGX leveraging OpenSSL, this solution offers less protection than the former but is contrasted by its versatility and better performance. These two solutions aims to guarantee the privacy of encryption keys with a threat model similar to the one of Intel SGX with a very strong adversary, which is able to arbitrarily run high privilege code. We assume the cryptographic primitives are implemented correctly on the chosen libraries. Additionally, we assume the attacker cannot break the security boundaries imposed by the secure enclave provided by Intel SGX, although we recognize that this might not always be the case in practice, as presented in section 2.3.1.

Finally, section 4.3 considers applications on which consistent and low execution times, as well as large quantities of memory, are required, making it unfeasible to deploy them under Intel SGX. We present a solution that monitors the application from a secure enclave to guarantee code and limited data integrity. Unlike the previous solutions, in this one we do not guarantee the privacy of the data, and assume that the protected application does not contain bugs that could lead to techniques not covered by our solution.

4.1 Apache's web server SSL module SGX integration

This section describes the modifications made to mod_ssl to make it compatible with Intel SGX in order to protect both the asymmetric private key and the negotiated symmetric key from unwanted third parties. This is achieved by moving the TLS implementation to the TEE, preventing access from both the operating system and the hypervisor. In addition, it also analyzes various cryptography libraries available to Intel SGX and identify that WolfSSL[47] is suitable for our needs and identifies a new issue on the TaLoS[48] library.

Initially, we evaluated two of the most popular web servers to integrate Intel SGX, NGINX and the Apache webserver. An initial analysis showed that the NGINX module that handled HTTPS has a much smaller codebase than the Apache's equivalent. However, the module's interaction with the cryptography library was through an API of NGINX. As we did not want to modify the source code of both the webserver and the module, we chose to modify the module for Apache as its cryptographic implementation is independent from the webserver.

4.1.1 Architecture

Our system should follow the typical architecture of any Intel SGX application containing both untrusted and trusted components, with the former querying the latter when work with sensitive data, is necessary.

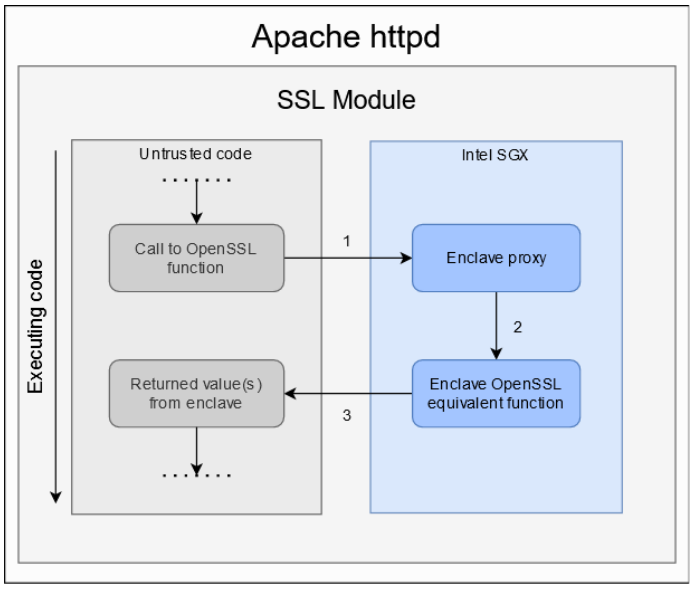


FIGURE 4.1: Proposed architecture for the Apache's http server.

Figure 4.1 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 figure 4.1), the enclave can utilize the private keys or other sensitive data in its possession along with a cryptography library (2 in figure 4.1) 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 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 figure 4.1) 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[49], we must implement a proxy that sanitizes the inputs received by the untrusted application seen in figure 4.1.

4.1.2 Cryptography library

Intel provides a fork of OpenSSL compatible with Intel SGX[50](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[48], 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, Tobias Cloosters et al. [49] 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[47], 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 read and writes in the enclave and maybe even code execution in certain cases[49].

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.

Appendix A.1 shows a very simple function in WolfSSL[47]. It returns the variable *rfd* in the pointer to a structure of type *WOLFSSL*. At 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 appendix 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 pointers. In our solution, we chose to create a HashMap for each type of structure used and exported to untrusted code. This allowed for a constant time access to the saved pointers regardless of the number of existing pointers.

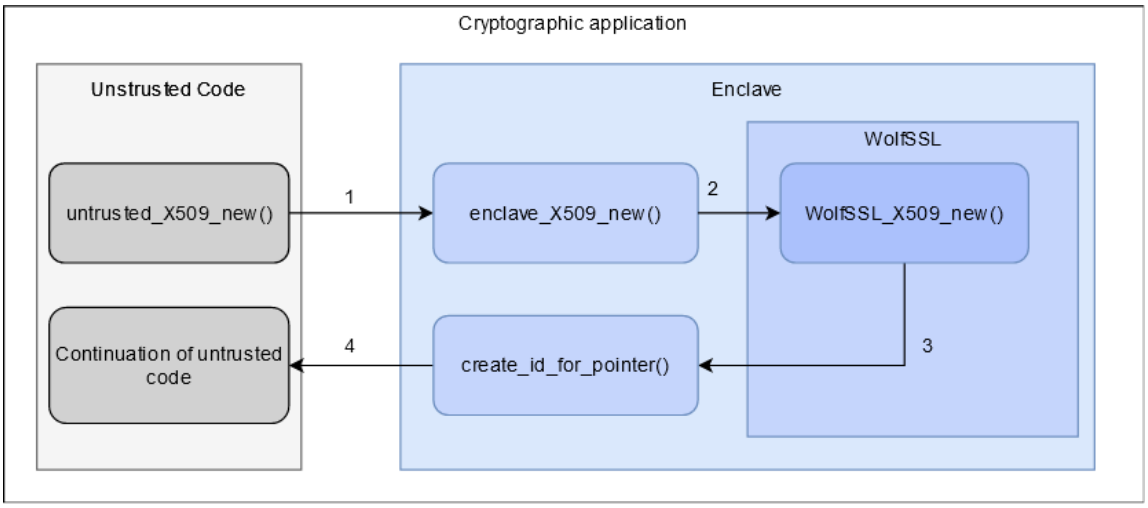


FIGURE 4.2: Preventing memory corruption in the enclave.

When the untrusted code calls a function in the enclave (1 in 4.2), the enclave will first check if any of the ids passed exist in the HashMap. If the passed ids contain a valid entry in the HashMap, the call is forwarded to WolfSSL (2 in figure 4.2). If the resulting function returns a non-NULL pointer, it is checked if it exists in the HashMap 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 HashMap with the corresponding pointer, and return the id to the untrusted code (4 in 4.2).

An HashSet could have been used in place of the HashMap, 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 HashMap would probably suffice in translating ids to pointer (i.e., ids to *void**), but we decided to create a HashMap for each type used within WolfSSl/OpenSSL. This allowed even for more 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 HashMap.

When the function to free the object is called, the corresponding id and pointers are removed from the corresponding HashMap. 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 HashMap will 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 the calls to OpenSSL functions. 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.

FIGURE 4.3: mod_ssl input chain

Figure 4.3 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 HashMaps, 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. Appendix 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 HashMap.

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[51], 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 subsection 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[48], even though it claims support for Apache's HTTP server, it does not treat 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 of the original module, we decided to have the possibility of loading normal keys and sealed keys by the enclave. We modified the module so that it would first try to load a sealed key, and if it failed, try to load the key normally. For convenience, and to 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. This property does not work too well with 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 that is present in TaLoS[48] 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[6] through inter-enclave communication.

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.

Enclave memory corruption

As mentioned in [49] 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. As this might be considered a software bug, this 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 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 but it is not a clear implementation in this solution, as mod_ssl uses the read and write functions of OpenSSL and we can not be sure when the key is used, only when

it is loaded. If an attacker finds a way to use the key through some of the exposed WolfSSL API that we did not cover, usage of the key by the attacker can go unnoticed.

4.1.7 Summary

In this section we described our port of *mod_ssl* to Intel SGX, allowing the utilized private key to be kept inside a secure enclave as well as the web server's TLS handshake. Although we wanted to keep the code as close to the original as possible we had to modify it to be compatible with the changes presented in section 4.1.4. Additionally, during the development of this solution we got a better understanding of the inner-workings of the OpenSSL library and discovered that OpenSSL Engines could be used to achieve a similar result in order to allow any application to take advantage of Intel SGX.

4.2 OpenSSL engine integration

As we will see in chapter 5 the previous solution comes with some big caveats and high-performance penalty, because of this 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 implementation differs from the previous solution as it allows for any application that uses OpenSSL as its cryptography library to leverage the keys protected by an enclave, guaranteeing its integrity and confidentiality, working very similarly to an hardware secure module or a pkcs#11 device. Unlike the previous solution it does not protect anything beside the utilized by the private key, meaning the established symmetric keys by applications in their TLS connections are vulnerable to the attacker.

4.2.1 Architecture

In this implementation the enclave works in a very similar manner to a pkcs#11 device. It exposes a limited number of functions so that untrusted code can load private keys and perform operations with it (i.e. Encryption and Decryption). Table 4.1 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.

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

TABLE 4.1: Exported function by the enclave to perform cryptography operations on a private key

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 of the Android Open Source Project[52] due to its simplicity. As the source code targets a version of OpenSSL prior to 1.1, with the OpenSSL Engine's help for pkcs#11 devices, libp11[53], we updated its source for a more recent version of OpenSSL.

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 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 figure 4.4, 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

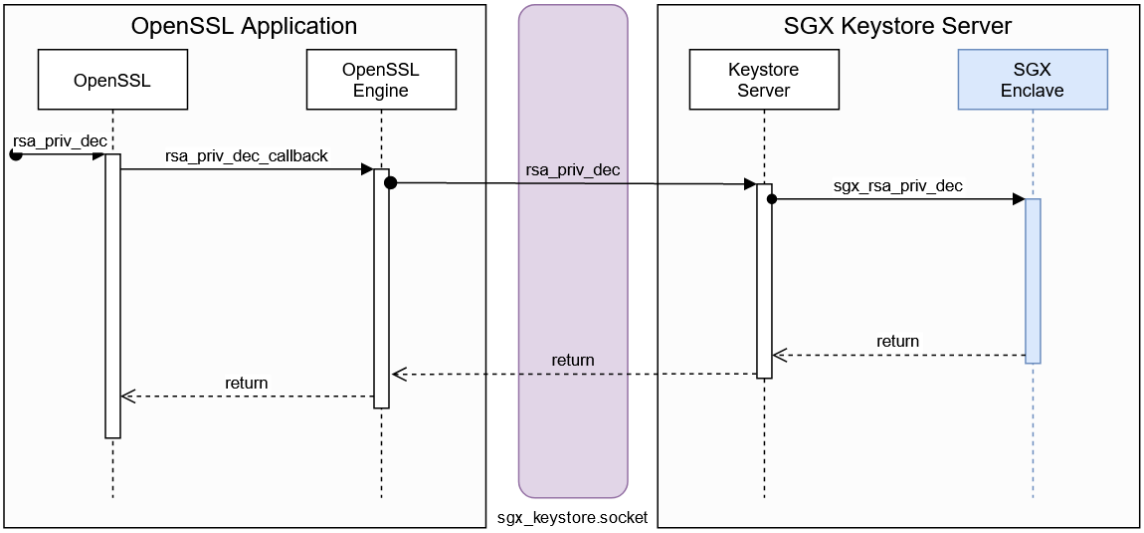


FIGURE 4.4: OpenSSL Engine with SGX key store architecture.

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.

4.2.2 Cryptography library

Unlike the implementation in section 4.1 the functionality required from the cryptography library is much smaller. The only requirements being the ability to load an RSA private key and perform operation with it. Fortunately, even though the functionality of Intel® SGX SSL[50] is rather limited, it provides support for the required features. We chose this library over WolfSSL[47] 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.

4.2.3 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, appendix 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 path specified in the configuration file. This should make it possible for any application to use the Engine, even from command like as shown in appendix B.3.

4.2.4 Required changes 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, returning 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. Eventually requesting it to load the key in the specified URI. For example, the URI *sgxkeystore:key.pem.sealed* would result in the initialization of the engine *sgxkeystore*, loading the specified URI. As the original code already supports pkcs#11 devices through libp11's OpenSSL engine[53], 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 appendix B.4.

4.2.5 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[54], 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 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, and this means that it is easily adaptable to network sockets so that future iterations could leverage a solution as presented in [35]. Allowing only authenticated access to the keys and logging key usage. Additionally because we only expose two functions that make operations with the key, we can easily implement logging on its usage.

4.2.6 Summary

Similarly to the previous solution, this keystore aims to protect the private key from being read by malicious parties but unlike the port of *mod_ssl* it accomplishes the same objective through a keystore leveraging SGX and integrating it with OpenSSL via an Engine. Unlike the ported module, this solution does not terminate the TLS connection inside the secure enclave. As we will see in chapter 5, the deployed engine contains some overhead over the unprotected solution but is not enough to affect the Apache web server under normal usage.

4.3 Secure Enclaves in Cheat Detection Hardening

This section gives the argumentation published in [1]. We explain how we employ secure enclaves to harden cheat detection mechanisms on modern AAA video games.

As not all applications are suitable to be executed inside a secure enclave due to the restrains one might impose, we implemented tamper detection techniques inside an enclave to detect modifications of an application that would otherwise not be protected.

Our implementation tries to be the least intrusive in the developer's life by minimizing the amount of the application's code that needs to be modified. It stores information about the application in a secure environment rather than moving the entire application to the enclave. This change allows for our solution to run efficiently inside the enclave as it does not exceed the physical memory limit imposed by Intel SGX.

The technique utilized to monitor the application works in a similar fashion as anti-cheat solutions seen in modern games, the implemented enclave acts as an extra layer of

security that will monitor changes to the application's code and data during runtime to guarantee that nothing has been tampered.

An overview of our solution is given in Figure 4.5, which shows it (Protection Oversight) working in parallel with the application's code while performing integrity checks on itself and the application's code. With the help of a kernel module, it achieves additional monitoring of the game processes. The communication between the Protection Oversight and the kernel module uses Netlink.

To see how solution affected the application's normal behavior we chose to monitor a fast-paced multiplayer game, Counter-Strike: Global Offensive, where input latency is critical and could be the deciding factor in a eSports tournament. Because any modifications to the game are detected, this also means that cheating opportunities are greatly reduced bringing fair-play to the game.

The proposed solution allows it to deploy on most native applications with no changes to its workings. It only requires modifications when the more generic type of checks does not catch application-specific attacks.

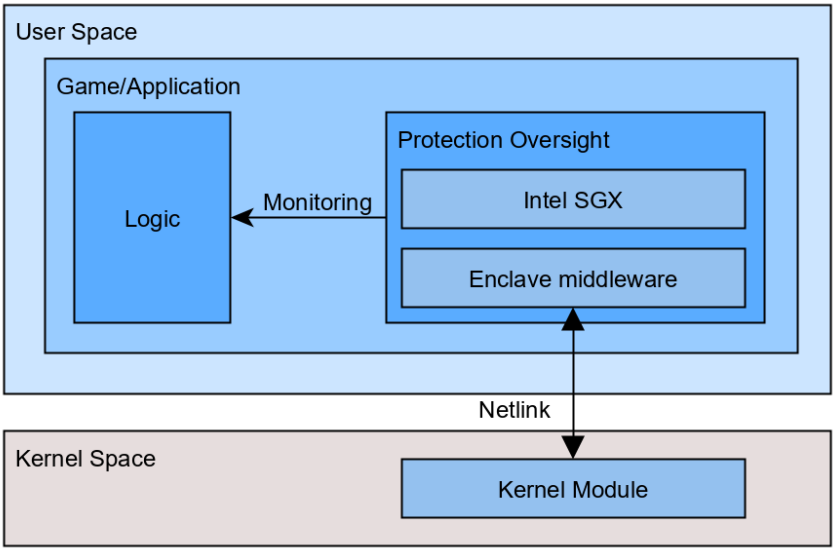


FIGURE 4.5: Implemented anti-cheat architecture

4.3.1 Enclave creation

In order to create the protecting enclave, we need to extract some information about the game, this process can be done during the release process of the game by the developer. This section defines the type of information needed and how it is obtained in our solution.

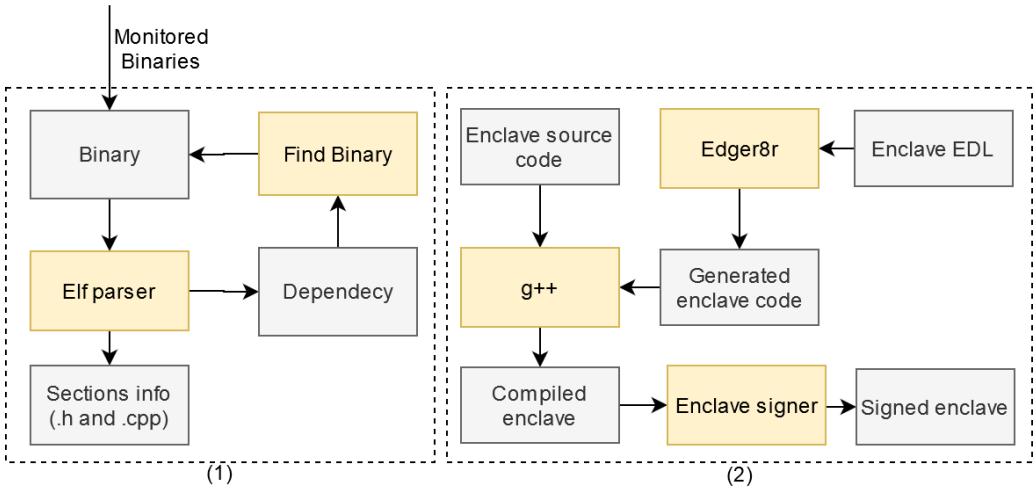


FIGURE 4.6: Enclave generation - Existing flow to generated enclave with sections to verify.

Analysis of the Protected Application

We need to guarantee that the enclave during runtime can read the memory regions of the game and its dependencies. This poses a problem as the section table of an ELF binary is not needed by the program during runtime. This information is copied to the enclave, during its creation, to circumvent this issue so that the enclave can access it. Additionally, we take advantage of this process (1 in Figure 4.6) to create hashes of the mapped sections and store them on the enclave. The information gathered is saved onto the enclave by generating code to compile along with the enclave's code (2 in Figure 4.6). This way, the enclave can, during runtime, monitor the sections in loaded memory with the expected ones using the hashes of each corresponding section.

The selection of the binaries that need protection is not a trivial task for two reasons. Firstly, users may have different versions of specific libraries, which will lead to different resulting hashes of the binary. Secondly, there may be legitimate reasons for the modification of a binary's read-only data [41]. Two examples of the latter include the library used to measure performance of the game[55], which detours methods within OpenGL so that it can measure the time between frames, and the popular open-source streaming software "Open Broadcaster Software" in "game capture" mode, which uses a detour to capture the generated frames.

4.3.2 Runtime Protection

After we have collected the information as mentioned earlier, the enclave can now perform, during runtime, the integrity checks on the binaries. A description follows of the necessary steps required for the enclave to be aware of the application's memory space.

Enclave Middleware Address Discovery

Since SGX provides a way to allocate memory on the stack in the untrusted environment, we can use this to find the stack pointer and frame pointer used by the application that invoked the enclave. With this information, we can calculate the offset at which the return address to the middleware is from the last known frame pointer with a disassembler's help.

With this data recovered, we can perform integrity checks on our solution (Protection Oversight in Figure 4.5) within the enclave.

Application Address Discovery

In an ideal world, the application would link its dependencies, including the enclave's middleware, statically. With this, the enclave could easily cover the game binary, as finding the middleware binary would imply finding the game binary. As statically linking every library is not practical, and since applications might load binaries through Operating System specific APIs (e.g. dlopen on Linux and LoadLibrary on Windows), parsing the Procedure Linkage Table (PLT) and the Global Offsets Table (GOT) sections of the loaded binaries might not yield all the required dependencies. As such alternative methods to find mapped binaries need to be used.

An alternative approach is to use TSX-based Address Probing (TAP) to discover the mapped memory regions in the game's process. TAP is a fault-resistant read primitive, introduced by Michael Schwarz et al. [56]. TAP leverages the Intel Transactional Synchronization Extensions (TSX) way of handling memory violations to stealthy query the device's virtual memory without raising a memory access violation on the kernel.

Unfortunately, in the same paper, it is mentioned that TAP takes approximately 45 minutes on an Intel i7-6700k to scan the entire virtual address space. This makes it impractical as it would require a substantial amount of time to complete while requiring a considerable amount of resources, which might affect the game's performance.

As it is impossible to modify the selected game's source code, the enclave needs to temporarily leave the trusted execution environment SGX to query the mapped regions' operating systems. This does give us the advantage that it is significantly faster to retrieve the mapped regions, but it is also a source that is not considered trustworthy by Intel SGX's threat model. It also comes at the advantage of being a more 'universal' solution since not all CPUs that support Intel SGX, necessarily support Intel TSX.

Code and Data Integrity Checks

Once the target binaries' addresses are acquired, the enclave can combine these with the information gathered in the first phase (1 in Figure 4.6) of the enclave's creation. The enclave can now monitor (as shown in Figure 4.5) the game's memory for any unwanted changes. While the read-only data section should be straightforward to check, game-specific values might require the developer to create additional checks within the enclave.

A common code interception technique in CS:GO cheats involves modifying the objects' pointer to the virtual function table, as these objects need to be modifiable. They cannot stay in read-only memory regions meaning that the checks, as mentioned earlier, do not cover them. The enclave must know where the objects are in memory, which must be specified by the developer, either by specifying a pointer path starting from a static address within the binary or by exporting symbols in the game code to allow the enclave to find the desired addresses, to protect against these types of attacks. This is ultimately the method used to detect the cheat tested during the evaluation of our results.

Kernel Space Aided Monitoring

Intel SGX is limited to run within user-space, limiting the surface area we can monitor. To circumvent this, as suggested by Figure 4.5, we load an optional kernel module that can communicate with the enclave middleware via Netlink. To monitor the application's interactions with the operating system, the kernel module takes advantage of the *ftrace* framework to intercept system calls in the kernel [57].

With the function *sys_open* intercepted, it is possible to monitor the file system's file accesses. To monitor a particular application, the enclave first requests, through the middleware, the kernel module to start monitoring. Afterward, the kernel obtains the context

of the process that wishes to be monitored and can start forwarding information to the enclave. This way, the enclave can be aware of all the file accesses the game makes and see if any additional files were loaded or modified. To reduce the number of events generated, we are strictly generating events for open requests with READ access (flags *O_RDONLY* or *O_RDWR*) to the file. The kernel module hashes the data and forwards the result along with the filename to the enclave to reduce the number of transitions between user space and kernel space instead of sending open events to the enclave.

To prevent external application from reading and to write in the game application, *process_vm_readv* and *process_vm_writev* are also analyzed to deny access to the game process memory. Additionally, access to the *ptrace* syscall is blocked when called against the game's process id to prevent the target application's debugging.

The enclave may also query the kernel for the allocated pages' permissions, as showed SGX does not provide enough mechanisms to know if the pages have execution permission.

4.3.3 Security Analysis

In this section, we provide a security analysis of the proposed system. We identify potential threats and how an attacker may attempt to exploit the system.

Integrity Checker

In a typical scenario, the code that verifies the game's code should be in theory as easily modifiable as the game's code, assuming no extra obfuscation has applied to it. In this proposal we have moved this code inside a secure enclave with Intel SGX, this gives us a strong hardware guarantees that an attacker has not modified the code running.

Game Code Modification

An attacker can get code execution within the game code in two scenarios: through the operating system's API; or finding an exploit in the game executable. Then it can modify everything within the game's memory region, including its code by altering the permissions of the allocated memory regions. To mitigate this, we perform integrity checks of the game during its runtime, and this means any modification made is visible to the enclave, and once the enclave is aware of it, it can take actions.

Malicious Hypervisor

As shown in SPIDER [58] by taking advantage of Intel's implementation of Second Level Address Translation, Intel Extended Page Table (EPT), the hypervisor can split code and data views as seen by the guest operating system. With this, SPIDER manages to place a breakpoint on the guest's memory while keeping this change invisible to the guest operating system. The same technique was showed by Satoshi Tanda [59] to place invisible inline hooks, which allows the modification of the execution flow of the guest's code to malicious code.

As stated by Intel, memory regions outside of enclaves are considered untrusted. If the attacker modifies the game's code using this method, when our solution attempts to read code sections of the game, the EPT will cause a data view of the address to be read and not the code view. This will lead our solution to think that nothing has changed when that is not the case.

Such scenarios are possible to detect by side-channel information[60, 60] that can be obtained by measuring the time taken to read or write to a memory region, as these will be measurably slower than unaffected memory regions. A different approach is to employ a solution that detects a hypervisor's presence and refuses to run under those circumstances. Igor Korkin[61] classifies four ways of identifying a hypervisor, signature-based, behavior-based, trusted hypervisor-based, and time-based.

Fault on Transitions To and From the Enclave

If already executing code inside the game, the attacker may attempt to mark the libraries responsible for transitioning to and from the enclave as non-executable and register an exception handler to handle these situations.

With this, the attacker can know when the transitions to and from the enclave occur. We believe that achieving this after the enclave has been loaded does not accomplish meaningful results to the attacker besides unloading the cheat's modifications on enclave entry and loading them on enclave exit. This attack involves continuously altering the page permissions and exception handling, which comes at a relatively high cost, enough to be influential in the game's performance, increasing the time each frame takes to render and thus lowering the frames per second. Depending on how often the enclave transitions to and from user space (i.e., how often checks are made to the game), the game's performance could drop to an unplayable state(less than ten frames per second). While it would

technically allow for cheating to work successfully, the player will have an unsatisfactory game experience because it will feel jerky and slow, making it impractical.

4.3.4 Summary

The work presented in this sections shows a design and implementation that aims to move cheat detection techniques to a secure enclave in Intel SGX, giving the developers a reliable guarantee that the attackers do not tamper the original code of the application. This is achieved by parsing the necessary information from the binary of the application in a secure machine during the creation of both the application and the enclave. The gathered information is later on, be used on the attacker's device to guarantee the integrity of the application.

Chapter 5

Results

In this chapter we present the performance tests made to our solutions and compare it to other existing solutions presented in chapter 3.

Our evaluation machine is 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 integrated graphics of the processor is used for the video output, an Intel Iris Pro Graphics 580. 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 [62] at commit b4673dc171fbe4e972bea4dc79aae17212bc29da.

Just like Graphene-SGX, SGX-LKL was obtained from its public GitHub repository[63] at commit a4fc0cc6fea39f30d33783e55626afbff3c7a871.

To utilize Scone, access to the community edition was requested and granted. There wasn't any kind of versioning besides the date on the docker images. The cross-compilers image from scone has digest 899ef9b2415bd2252c8a3ce396599cc957405f9c9333f6b7d39d95fe98fc00f2.

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 loads 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 big amount of transitions and exceeds the amount of physical memory available.

5.1.2 Results

In figure 5.1 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 seconds. This first value of this solution is not represented in the graph because we chose to limit the vertical axis of the graph to 5 seconds 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 decrease, so does the execution time until 1 MiB, buffers higher than that resulted in an increase of 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[6] 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[4] 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.

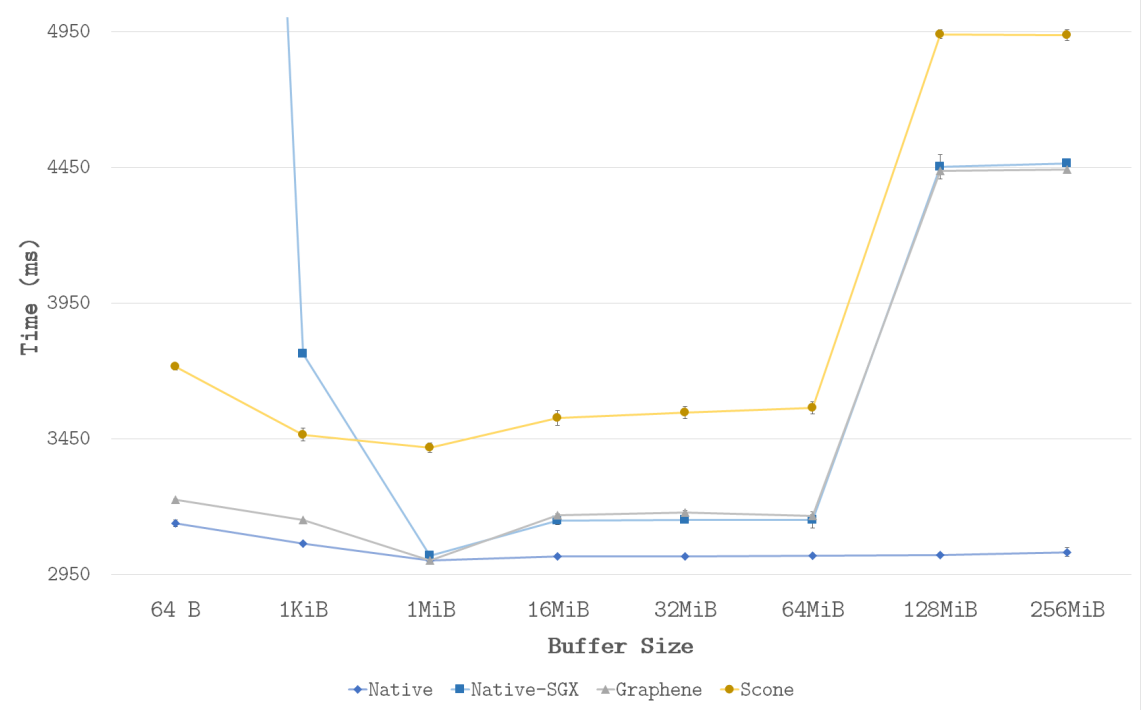


FIGURE 5.1: Average sigma time taken to hash 256 MiB of random data on disk with different buffer sizes.

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[6], SGX-LKL[7] and Scone[4].

5.2.2 Results

In table 5.1 we can see our solution described in section 4.2 identified by ”Engine” 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 is where 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 the Alexa's list of the most popular sites in the world, is 2048 bit[64], on which our solution performs 15.07% worse than the native solution without any protection. Verification of signatures are unaffected as it is an operation which 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

Solution \ Key size	512	1024	2048	3072	4096
Native	23286 ± 171	10646 ± 59	1685 ± 3	545 ± 1	252 ± 0
Engine	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

TABLE 5.1: RSA signatures per second on different solutions and various key sizes

see that our solution from the solution that utilizes SGX performs significantly worse up until RSA key sizes of 2048 bit. 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 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 an 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 web server 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.

To measure each solution's impact on the web server, we utilize ApacheBench[65] 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, with the exception of the last test which 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 figure 5.2 we can see both of our solutions compared to the normal Apache web server running normally and with Graphene-SGX[6]. We can see that our solution utilizing the custom OpenSSL Engine (Engine-SGX) 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 5.1 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[6]

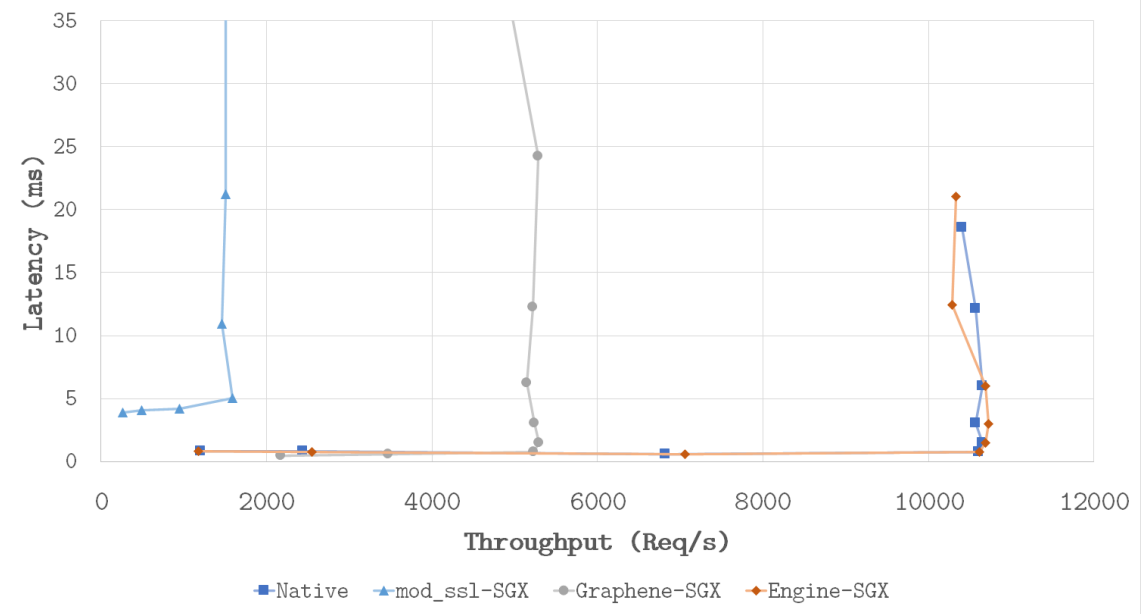


FIGURE 5.2: Throughput versus latency of Apache's web server workload. The lower and further right the better

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 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 web server when requiring calls to the cryptography API. If we take the example of figure 4.3 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 Graphne-SGX, this input chain would at most, 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, which might have caused 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 web server 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 that it accepted the TCP connections, it did not reply with the contents of the page, and the connection just hanged.

Although Scone[4] benchmarked Apache's web server, the current solution[31] does not provide a curated imaged for the Apache's web server. Compiling the web server from source and executing it inside Scone results in the same behavior as SGX-LKL, the

connection hanged.

Our test does not include TaLoS[48] as we could not compile the web server against it due to missing functions in the cryptography library when built in hardware mode.

5.4 Cheat Detection Hardening

The results presented in this section have been published in [1].

In this section, we show the test case used during the evaluation of our solution when applying SGX to an Anti-Cheat engine and show the obtained results. For this, we measured two metrics: The first test is the overhead associated with our solutions in the loading of the game; the second is the gameplay overhead associated with the middleware running. To test this, we ran the tests ten times with and without our implementations for each of the options.

During the game benchmark we measured the time taken to render each frame, also called the frame time, using *libperflogger* [55] so that we could take the average frame time of all the frames, the average frame time of the worst 1% and the average frame time of the worst 0.1%. With these metrics, we can see how bad the game performs in its worst-case scenarios, where a small number of frames take a relatively big time to render.

5.4.1 Methodology

Our game of choice to test our solution was Counter-Strike: Global Offensive along with the Open Source Linux cheat Fuzion [66]. To consistently measure the game's performance, with and without our solution, we use an FPS Benchmark available in the steam workshop [67].

During our testing, the game used between 1 and 1.5 GB of memory. We have configured our enclave with a maximum stack and heap sizes to 256 kilobytes and 1 megabyte, respectively. As the generated binary's size is 862 kilobytes, we can conclude that our solution uses at most roughly 2.1 megabytes of the Intel SGX allocated memory.

As we used the CPU's integrated graphic card, we decided to run the game at a resolution of 1920:1080 with the lowest possible quality settings with smoke grenades disabled. This is because we wanted the frame time to be CPU-bound as possible to test the impact of our solution.

We chose to monitor the game's executable, its *engine_client.so* dependency along with the array of global objects present in the exported symbol *s_pInterfaceRegs*, the enclave's middleware, and the SGX's dependencies. While it was possible to monitor all the dependencies of the game binary, it led to some false positives mainly because the library used to measure the game's performance, which, in order to do so, must intercept OpenGL functions. To counter this, we identified the binaries with functions of interest for the cheaters.

5.4.2 Results

Our enclave took on average 161.60 ms $\pm$ 45.05 ms to load while only marginally increasing the time taken to load the game from 23.75 s $\pm$ 0.42 s to 24.12 s $\pm$ 0.38 s.

Frame time	No Protection	Every second	Every 10 seconds
Avg (ms)	8.52 ± 2.38	8.60 ± 2.46	8.57 ± 2.40
Avg worst 1% (ms)	14.39 ± 1.08	14.75 ± 1.20	14.51 ± 1.12
Avg worst 0.1% (ms)	16.98 ± 1.08	17.56 ± 0.99	17.29 ± 1.10

TABLE 5.2: Comparison of the time each frame took, on average, to render

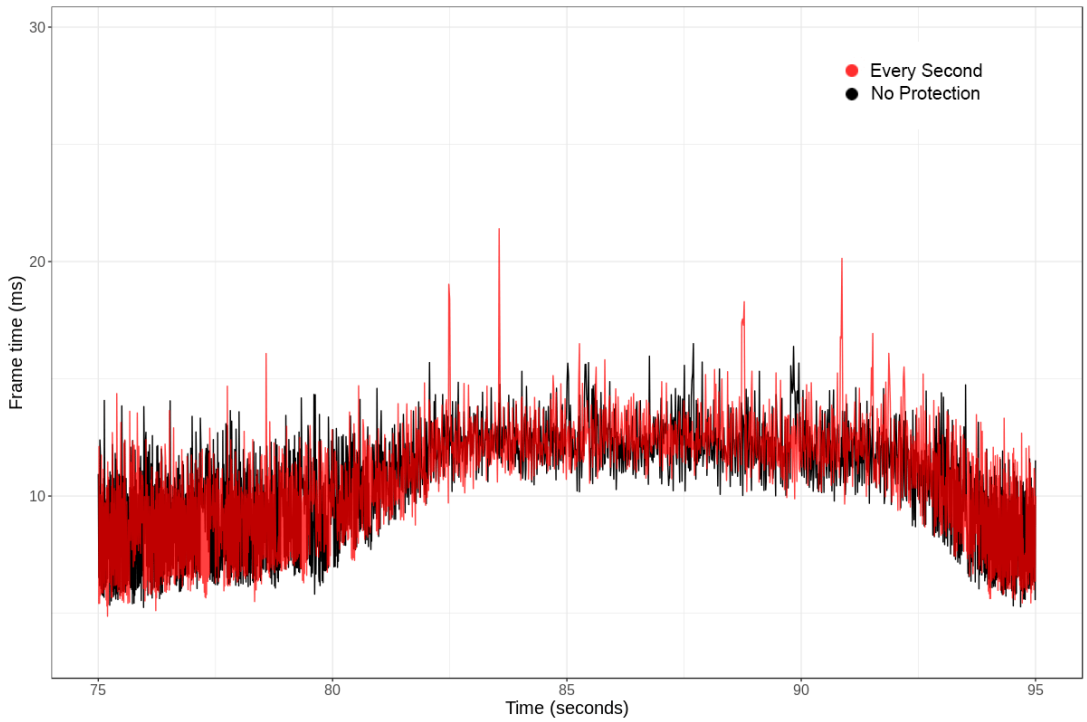


FIGURE 5.3: A time interval of 20 seconds, with the frames per seconds of our solutions compared with the baseline

When looking at how the game performs, it is not enough to look at all the frames' average frame time. It is essential to identify moments when frames take a relatively long time to render as these may cause sudden, short freezes during the gameplay. From the average frame time of all frames in table 5.2, it appears that our solution has a shallow impact on the gameplay's performance, while only marginally increasing the frame times of the worst 1% and 0.1% by 2.50% and 3.42% respectively. In Figure 5.3, we present a 20-second portion of the benchmark. This allows us to see the increase in frame time when our solution runs, and we can identify this by the sudden peaks in the graph.

When the enclave is not idle, i.e., it is performing the monitoring of the game code. We can detect the cheat Fuzion [66] successfully within $31.62\,\text{ms} \pm 2.47\,\text{ms}$. This time is dependent on the number of sections we are verifying. On average 5MB of data takes $13.55\,\text{ms} \pm 0.58\,\text{ms}$ to verify.

Chapter 6

Conclusion and future work

In this chapter to overview our work and results accomplished throughout this thesis. First, we make a brief overview of the research and development made during the thesis, commenting on the goals achieved and complexity of each project. Secondly, we talk about the results obtained, the effectiveness of the developed and their viability. Finally, we also hint at possible future iterations of this work so that the utilization of Intel SGX secure enclaves impacts significantly less the applications used during this thesis.

6.1 Research and development

During the research phase of this thesis, we struggled to find applications that utilized Intel SGX, instead we found many solutions that either automatically porting applications[3] or ran unmodified applications inside Intel SGX[4–7]. 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.

Because of the aforementioned situation, we decided to port some applications to utilize Intel SGX through the SGX SDK and compare how these would perform relative to their original counterparts running without Intel SGX and running with the previously mentioned solutions.

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 most of the developing time of this thesis. We encountered many issues, from finding a compatible library with it and Intel SGX, to modifying the cryptography library so that we could use it from untrusted code and through its OpenSSL compatibility layer.

Developing the Keystore for Intel SGX took significantly less time than porting *mod_ssl*, being the harder task of this solution creating the OpenSSL Engine as there were lack of simple or properly documented implementations.

We also concluded that not every application cannot reliably be executed within an enclave due to the constrains it imposes on the application. To overcome this, we successfully applied anti-cheat monitoring techniques to a secure enclave to protect a game from unwanted modification. The test case for this solution was Counter-Strike: Global Offensive, a closed-source first-person shooter created by Valve.

Additionally, in section 4.1 we also identified some issues with the Intel SGX cryptography library, TaLoS, and explained why it would not be suitable for production.

6.2 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 in order to protect both the private key and the generated symmetric key during the handshake. This protection however comes with a great performance penalty when 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 to guarantee secrecy of private keys, involved implementing a keystore in Intel SGX and integrating with an OpenSSL engine so that applications could transparently use it. When applying this solution to the Apache web server it saw no measurable overhead compared to the original solution and performed 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 5.1 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 were unable to 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 the versatility and performance the second option it is hard to recommend the first implementation which not only has an inferior performance but 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.

When applying cheat detection techniques to Intel SGX we noticed a measurable overhead over the base game, but we concluded that it did not significantly impact the game's performance. We were able to successfully monitor the game's code and data for changes, which meant we successfully stopped cheating opportunities without requiring to move the entire game to a secure enclave. During the testing, we would like to have had a better dedicated GPU in the test machine so that we could better analyze it in a real world scenario, which would probably lead to a better results as a dedicated GPU does not have to share memory bandwidth with the CPU unlike the utilized integrated CPU. Additionally, this work resulted in a publication in the 17th International Conference on Trust, Privacy and Security in Digital Business, TrustBus2020.

6.3 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 out throughout the thesis. Therefore, we would like for future iterations of this work to overcome some of 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[48] has showed that the implementation of asynchronous has improved performance on their workload by has much as 117%. In section 5.1, we can see that other solutions like Scone[4] and [6] which implemented asynchronous function calls perform significantly better than our solution when an high amount of enclave transitions are made. To mitigate some of the performance penalty of our solutions, we would like for future iterations of our work to implement asynchronous function calls.

Public-key cryptography support

Due to time constrains, 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.

Enhancing anti-cheat engine

The presented solution requires the executable game information to be present within the enclave binary, future implementation could take Intel remote attestation service to guarantee a secure communication with game servers to load this information and also to inform if a player is cheating.

Due to the limitations imposed by Intel SGX, monitoring a program outside an enclave is not easy and described in Subsection 4.3.3 that our solution might be prone to some attacks that are undetectable in the current implementation. We are aware that communication between the kernel module and the enclave can be intercepted and tampered, but this attack will only stop detecting some new cheats that use techniques unknown to the enclave.

In the future work, we want to look at other types of games, measuring how adequate our solution is to each one of them. It can also be interesting for our solution to integrate with SGXElide [19] to enable code secrecy of the enclave's code, making it so the attacker can only see the enclave as a true black-box.

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.

A.1 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.

A.2 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 thesis.

B.1 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. 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 which will ultimately call the registered function.

B.2 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.

B.3 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.

B.4 Patch to support another engine on Apache web server

```
--- ssl_util.c      2020-07-16 09:28:12.084116397 +0100
+++ ssl_util.c.sgxkeystore  2020-07-16 09:27:25.444073743 +0100
@@ -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, "pkcs11:", 7) == 0;
+    return strncmp(name, "pkcs11:", 7) == 0
+           || strncmp(name, "sgxkeystore:", 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.

Bibliography

[1] A. Brandão, J. Resende, and R. Martins, "Employment of secure enclaves in cheat detection hardening," in *The 17th International Conference on Trust, Privacy and Security in Digital Business - TrustBus2020*, 2020.

[2] C. Fontaine and F. Galand, "A survey of homomorphic encryption for nonspecialists," *EURASIP Journal on Information Security*, vol. 2007, pp. 1–10, 2007.

[3] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for intel {SGX}," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 285–298.

[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 689–703.

[5] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[6] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 645–658.

[7] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," *arXiv preprint arXiv:1908.11143*, 2019.

[8] J. Yan and B. Randell, "A systematic classification of cheating in online games," in *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, 2005, pp. 1–9.

[9] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 1*, vol. 1, 2011.

[10] C. Domas, "The memory sinkhole," *BlackHat USA*, 2015.

[11] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[12] J. P. Buzen and U. O. Gagliardi, "The evolution of virtual machine architecture," in *Proceedings of the June 4-8, 1973, national computer conference and exposition*. ACM, 1973, pp. 291–299.

[13] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.

[14] "Globalplatform technology tee system architecture version 1.2," https://globalplatform.org/specs-library/?filter-committee=tee, accessed: 2020-01-12.

[15] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[16] Intel, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 3B, 2011.

[17] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 523–539.

[18] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 440–457.

[19] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "Sgxelide: enabling enclave code secrecy via self-modification," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 75–86.

[20] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 38–54.

[21] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 523–539. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk

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

[23] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018, see also technical report Foreshadow-NG [68].

[24] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading kernel writes from user space," *arXiv preprint arXiv:1905.12701*, 2019.

[25] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[26] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," 2020.

[27] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," https://sgaxeattack.com/, 2020.

[28] "Openenclave github repository," Set 2020. [Online]. Available: https://github.com/openenclave/openenclave

[29] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 291–304.

[30] "musl libc," 2019. [Online]. Available: https://www.musl-libc.org/

[31] https://scontain.com/index.html. Scone - a secure container environment. [Online]. Available: https://scontain.com/index.html

[32] J. A. Donenfeld, "Wireguard: Next generation kernel network tunnel." in *NDSS*, 2017.

[33] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "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.

[34] K. Krawiecka, A. Kurnikov, A. Paverd, M. Mannan, and N. Asokan, "Safekeeper: Protecting web passwords using trusted execution environments," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 349–358.

[35] A. Kurnikov, A. Paverd, M. Mannan, and N. Asokan, "Keys in the clouds: Auditable multi-device access to cryptographic credentials," *CoRR*, vol. abs/1804.08569, 2018. [Online]. Available: http://arxiv.org/abs/1804.08569

[36] D. Tian, J. I. Choi, G. Hernandez, P. Traynor, and K. R. Butler, "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, pp. 255–266.

[37] L. Richter, J. Götzfried, and T. Müller, "Isolating operating system components with intel sgx," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, 2016, pp. 1–6.

[38] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Ndss*, vol. 3, no. 2003. Citeseer, 2003, pp. 191–206.

[39] "Bitdefender hypervisor introspection (hvi) security solution." [Online]. Available: https://www.bitdefender.com/business/enterprise-products/hypervisor-introspection.html

[40] E. Bauman and Z. Lin, "A case for protecting computer games with sgx," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, 2016, pp. 1–6.

[41] J. Berdajs and Z. Bosnić, "Extending applications using an advanced approach to dll injection and api hooking," *Software: Practice and Experience*, vol. 40, no. 7, pp. 567–584, 2010.

[42] W.-c. Feng, E. Kaiser, and T. Schluessler, "Stealth measurements for cheat detection in on-line games," in *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, 2008, pp. 15–20.

[43] B. Innovations. Battleye anticheat. [Online]. Available: https://www.battleye.com/

[44] D. Cowley, "Epic games acquires kamu, game security and player services company," *Unreal Engine Blog*, oct 2018, https://www.unrealengine.com/en-US/blog/epic-games-acquires-kamu-game-security-and-player-services-company.

[45] I. Epic Games. Easy anti-cheat. [Online]. Available: https://www.easy.ac

[46] L. GameBlocks. Fairfight server-sided anti-cheat. [Online]. Available: https://gameblocks.com/

[47] WolfSSL, "Wolfssl," https://www.wolfssl.com/, online; Acessed on 07-Maio-2020.

[48] P.-L. Aublin, F. Kelbert, D. O'keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "Talos: Secure and transparent tls termination inside sgx enclaves," *Imperial College London, Tech. Rep*, vol. 5, p. 2017, 2017.

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

[50] Intel, "Intel® software guard extensions ssl library," *arXiv preprint arXiv:1908.11143*, 2017.

[51] OpenSSL, "Openssl 1.1.0 changes." [Online]. Available: https://wiki.openssl.org/index.php/OpenSSL_1.1.0_Changes

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

[53] OpenSC, "libp11," https://github.com/OpenSC/libp11, 2020.

[54] OpenDNSSEC. Softhsm. [Online]. Available: https://www.opendnssec.org/softhsm/

[55] Lurkki14, "libperflogger - game performance logging library," https://github.com/Lurkki14/libperflogger, 2019.

[56] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel sgx," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 177–196.

[57] S. Rosted., "Ftrace kernel hooks: More than just tracing." *Linux Plumbers Conf*, 2014.

[58] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 289–298.

[59] S. Tanda, "Ddimon," https://github.com/tandasat/DdiMon, 2018.

[60] I. Kyte, P. Zavarsky, D. Lindskog, and R. Ruhl, "Enhanced side-channel analysis method to detect hardware virtualization based rootkits," in *World Congress on Internet Security (WorldCIS-2012)*. IEEE, 2012, pp. 192–201.

[61] I. Korkin, "Two challenges of stealthy hypervisors detection: time cheating and data fluctuations," *arXiv preprint arXiv:1506.04131*, 2015.

[62] Oscarlab, "Graphene github repository," Dec 2019. [Online]. Available: https://github.com/oscarlab/graphene

[63] "Sgx-lkl github repository," Nov 2019. [Online]. Available: https://github.com/lsds/sgx-lkl/

[64] I. Qualys. (2020, jul) Ssl pulse. [Online]. Available: https://www.ssllabs.com/ssl-pulse/

[65] Apache. Apache http benchmarking tool. [Online]. Available: http://httpd.apache.org/docs/2.4/programs/ab.html

[66] LWSS, "Fuzion," https://github.com/LWSS/Fuzion/, 2019.

[67] Counter strike: Global offensive benchmark map. [Online]. Available: https://steamcommunity.com/sharedfiles/filedetails/?id=500334237

[68] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018, see also USENIX Security paper Foreshadow [23].