Trusted Reference Monitors for Linux using Intel SGX Enclaves

Alexander Harri Bell-Thomas
Jesus College

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the degree of
Master of Engineering in Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: Alexander.Bell-Thomas@cl.cam.ac.uk

June 4, 2020

Declaration

I, Alexander Harri Bell-Thomas of Jesus College, being a candidate the Part III of the Computer Science Tripos, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 0

Signed:

Date:

This dissertation is copyright © 2020 Alexander Harri Bell-Thomas.
All trademarks used in this dissertation are hereby acknowledged.

Abstract

Write a summary of the whole thing. Make sure it fits in one page.

Contents

1 Introduction **1**

2 Background **5**

2.1 Intel SGX 5
    2.1.1 Security Characteristics 6
    2.1.2 Architecture and Implementation 7
    2.1.3 Enclave Lifecycle 9
    2.1.4 Attestation 10
    2.1.5 Provisioning and Sealing 10
2.2 Modern *libOSes* 10
2.3 Information Flow Control 10
2.4 Aspects of the *Linux* Kernel 10

3 Related Work **11**

4 Design and Implementation **13**

5 Evaluation **15**

6 Summary and Conclusions **17**

List of Figures

2.1 Abstract overview of SGX's protection in an adversarial environment. . 6
2.2 A high-level overview of the SGX hardware and software architecture. 7

4.1 Abstract overview of an SGX enclave's protections. 14

List of Tables

2.1 Overview of notable SGX x86 instructions in an enclave's lifecycle. . . 9

Chapter 1

Introduction

The task of defending computer systems against malicious programs and affording isolation to protected system components has always been exceedingly challenging to achieve. A system's *Trusted Computing Base*, or *TCB*, defines the minimal set of software, fimrware, and hardware components critical to establish and maintain system security and integrity. This traditionally includes, amongst others; the OS kernel; device drivers; device firmware; and the hardware itself. Compromise of a trusted component inside a system's *TCB* is a direct threat to any secure application running on it. A common approach to hardening a system's security is to minimise its *TCB*, diminishing its potential *attack surface*.

A modern trend is to outsource the physical layer of a system to a foreign party, for example a *cloud provider* — this is beneficial both in terms of cost and flexibility, but confuses many security considerations which assume that the physical layer itself can be trusted. In this context there is no guarantee of this, as the physical layer is usually provided as a *virtual machine*, inflating the system's *TCB* with an external and transparent software layer, the underlying *hypervisor*.

The concept of *Trusted Execution Environments*, *TEEs*, has been explored by the security community for a very long time as potential protection against this, providing isolated processing contexts in which an operation can be securely executed irrespective of the rest of the system — one such example is software *enclaves*. *Enclaves* are general-purpose *TEEs* provided by a CPU itself, protecting the logic found inside at the architectural level. Intel's Software Guard Extensions (SGX) is the most prolific example of a *TEE*, affording a *black-box* environment and runtime for arbitrary apps to execute under. Introduced by Intel's *Skylake* architecture, a partial view of the platform's working can be found in whitepapers and previous publications.

SGX provides a *TEE* by enforcing the following:

1. Isolating a protected application, coined an enclave, from all other processes on the system at any priviledge level using hardware enforcement.
2. Protecting reserved memory against attacks using a dedicated hardware component, the *Memory Encryption Engine* (*MEE*).

... bit more here, talk about attestation and measurement briefly

A common usage pattern for enclaves in modern production systems is to build on top of a *libOS*.[1] This approach sees a trusted application built to depend on a modified operating system which is loaded alongside it in the enclave. Examples include *SGX-LKL*, *Graphene*, and *Occlum*. These projects allow SGX-unaware applications to be inexpensively ported into enclaves, but drastically inflate the *TCB* of the resulting program.

The aim of this work is to explore methods of hardening Linux with an SGX-driven *reference monitor* to track and protect host OS system resources using *information flow control* methods. Further, it aims to investigate whether bundling an entire operating environment into an enclave is a necessity, instead asking if simply using the host operating system, once hardened, would suffice in some situations.

Our Contribution

This work provides:

---

[1] Library Operating System

- A prototype implementation of an enclave-based, modular *reference monitor*, empowering *information flow control* techniques to operate with autonomy and protection from the host operating system. Enforcement is achieved using a modified Linux kernel, with an overall *TCB* including only a minimal footprint of the core kernel alongside the enclave application.
- A userspace interposition library to near-transparently integrate unmodified applications to fully function under the new restrictions.
- A full port of the *libtomcrypt* cryptography library for use inside an SGX enclave.
- A rigorous investigation of the performance implications of this approach, featuring a lightly-modified version of the *Nginx* production webserver. Worst-case performance shows a 35% decrease in request throughput, with the common case reporting 7-11%. Additionally we report a median overhead of 39μs (IQR 26-72μs, $n = 10^6$) per affected *system call*, matching or surpassing similar, non enclave-based, systems.

Chapter 2

Background

This chapter will cover a number of topics essential to understanding the rationale and implementation of the design as discussed in § 4. These include; Intel SGX, a brief overview of modern *libOSes*, an introduction to *Information Flow Control* (*IFC*), and a overview of key aspects of the Linux kernel relevant to the architecture of the prototype.

2.1 Intel SGX

Intel's Software Guard Extensions, SGX, was first announced and detailed in a handful of whitepaper documents published in 2013. [X,Y,Z] It described a novel approach, creating in-CPU containers with their own protected memory pools. These regions, called *enclaves*, cannot be read from or written due to fundamental protection mechanisms provided by the x86 architecture. *Enclaves* provide both integrity and secrecy to the operation running inside of it, even in the prescence of a malicious host.

Motivation At a high level SGX aims to achieve security for sensitive application by shielding them, and the resources it uses, against tampering and to provide a guarantee to end users about an enclave's integrity; this is achieved using measurement and attestation. A driving use case is in a cloud computing context, where users are forced to trust a foreign party with both their data and business logic. By distributing

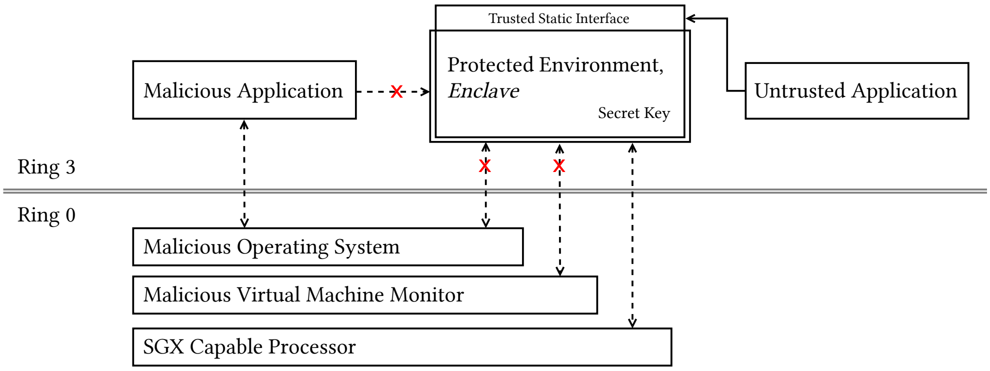
Trusted Static Interface
Protected Environment,
Enclave
Secret Key
Malicious Application
Untrusted Application
Ring 3
Ring 0
Malicious Operating System
Malicious Virtual Machine Monitor
SGX Capable Processor

Figure 2.1: Abstract overview of SGX's protection in an adversarial environment.

encrypted, yet executable, containers targetting a single, unique SGX CPU, users can be assured that their information is safe, regardless of any virutalisation that may be taking place. Only the provisioned CPU is able to decrypt and execute the enclave, strictly in accordance with the restrictions of the SGX platform.

2.1.1 Security Characteristics

At its heart SGX is designed to be *trustworthy*; this is achieved in a number of ways, including robust enclaving provisioning, sealing and attestation. Intel enumerates SGX's protections as follows;

- Memory security against observation and modification from outside the enclave; this is achieved using an in-die *Memory Encryption Engine* (*MEE*), with a secret that rotates on every boot. This protection notably works against a host hypervisor, other enclaves, and anything running in supervisor mode.
- Attestation of an enclave to a challenger through the use of a permanent hardware security key for asymmetric encryption.
- Proxied software calls to prepare and transfer control in and out of an enclave. Arguments are securely marshalled according to a static enclave definition.
- SGX does not defend against reverse engineering or sidechannel attacks: [X,Y] this is the responsibility of the developer to mitigate.
- Debugging support is only provided via a specialised tool and only when an

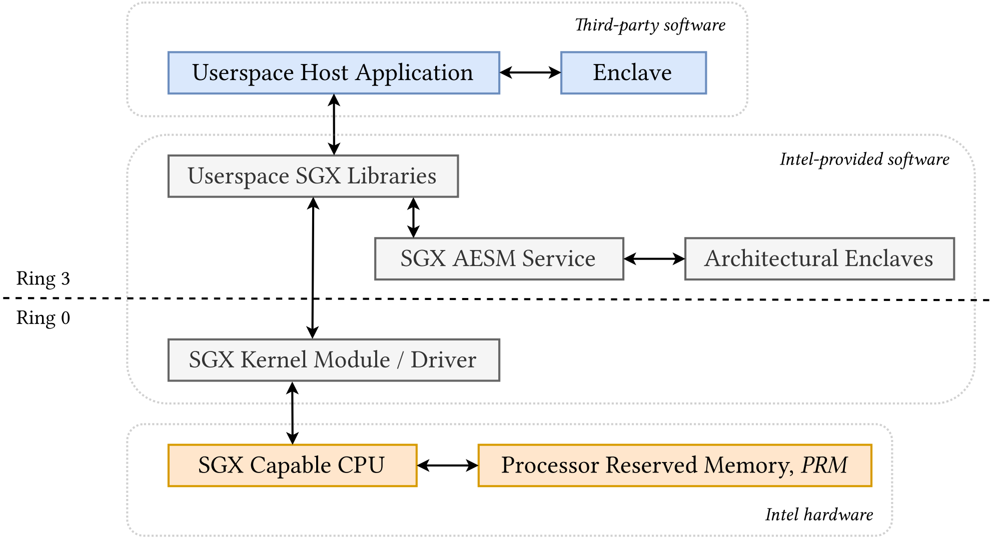
Third-party software
Userspace Host Application
Enclave
Intel-provided software
Userspace SGX Libraries
SGX AESM Service
Architectural Enclaves
Ring 3
Ring 0
SGX Kernel Module / Driver
SGX Capable CPU
Processor Reserved Memory, *PRM*
Intel hardware

Figure 2.2: A high-level overview of the SGX hardware and software architecture.

enclave is compiled with debugging enabled.

2.1.2 Architecture and Implementation

The SGX platform comprises a number of interlocking parts, as shown in Figure 2.2. Working from the hardware up, at the heart of the platform is the extended x86 instruction set and memory protection provided by an SGX-capable CPU.

Hardware Enclaves' data and code is stored securely in *Processor Reserved Memory, PRM*; this is a set of pages in system memory that are presided over by the *MEE*. DMA[1] to *PRM* is always rejected. *PRM* consists of two datastructures; the *Enclave Page Cache Map* (*EPCM*) and the *Enclave Page Cache* (*EPC*). An individual enclave is defined by an *SGX Enclave Control Structure, SECS*; this is generated when an enclave is created and stored in a dedicated entry in the *EPC*. An enclave's *SECS* contains important information such as its (system) global identifier, its measurement hash and the amount of memory it is using. Access control information is stored in the *EPCM* alongside page validity flags, the owning enclave identifier and the page's type; this is not accessible from software. An attempt to resolve a page in *PRM* is successful only if the CPU is executing in enclave mode and its *EPCM* entry states it belongs the currently executing enclave — if this is not the case the lookup returns an unused page from generic system memory.

---

[1] Direct Memory Access

The host OS or hypervisor manages the *EPC* just as it does with normal system memory, swapping pages in and out according to its own policy, but must do so using SGX specific instructions. The *MEE* is responsible for ensuring the integirty and confidentiality of this process, encrypting and decrypting pages as they cross the *PRM* boundary. Data is verified with the use of an integrity tree, and encryption keys are generated at boot-time. Importantly the SGX archicture relies on the host OS being SGX-aware, empowering userspace applications to function without privilege; this is provided by the SGX driver, *isgx*.

Userspace services Starting an enclave requires retrieving a *launch token* from Intel's *Launch Enclave*; this checks the signature and identity of the enclave to ensure it is valid. Access to the *Launch Enclave* and other architectural enclaves is provided by the AESM service; the userspace SGX libraries facilitate the communication mechanism. Other architectural enclaves include;

- The *Provisioning Enclave* — this verifies the authenticity of the platform and retrieves an enclave's *attestation key* from the *Intel Provisioning Service's* servers.
- The *Quoting Enclave* — this provides trust in the identity of the SGX environment and enclave being attested, by converting the locally generated *attestation key* to a remotely-verifiable *quote*.

Third-party enclaves Enclaves can only be entered via userspace, as detailed in § 2.1.3, but are always accompanied by a host application which acts as its untrusted counterpart. The host application calls the SGX SDK to build an enclave on its behalf using an enclave image, packaged as a standard shared library (`enclave.so`) and returns its *global identifier*. Control is passed from the host application to the enclave by invoking an enclave function via an *ECALL*. Execution flow can temporarily leave the enclave if it call one of the host application's function via an *OCALL*. Execution naturally leaves enclave-mode when the *ECALL* terminates. Both *ECALLs* and *OCALLs* are

Execution Mode	Instruction	Function
Ring 0	`ECREATE`	Generate and copy the *SECS* structure to a new page in the *EPC*, initialising a new enclave.
	`EADD`	Add a new *EPC* page for the current enclave; this is used to load initial code and data.
	`EEXTEND`	Updates the enclave's measurement during attestation; modifies the *SECS*.
	`EINIT`	The terminal instruction in an enclave's initialisation, finalising its attributes and measurement.
	`EREMOVE`	Permanently remove a page from the *EPC*; usually invoked during enclave destruction.
Ring 3	`EENTER`	Transfer control from the host application to a pre-determined location in an enclave.
	`ERESUME`	Re-enter the enclave after an *OCALL* and resume execution.
	`EEXIT`	Restore the original operating mode at the location `EENTER` was triggered and flush the TLB.
	`EGETKEY`	Access platform cryptography keys required for attestation and sealing.
	`EREPORT`	Generate a *report* for an enclave's *attestation key* for an attestation process.

Table 2.1: Overview of notable SGX x86 instructions in an enclave's lifecycle.

defined statically in the enclave's interface definition (`enclave.edl`). The necessary glue code is generated by the SGX SDK's build toolchain at compile time; this ensures calls crossing the enclave boundary are marshalled safely.

2.1.3 Enclave Lifecycle

Something interesting here

2.1.4 Attestation

2.1.5 Provisioning and Sealing

2.2 Modern *libOSes*

2.3 Information Flow Control

2.4 Aspects of the *Linux* Kernel

Chapter 3

Related Work

Chapter 4

Design and Implementation

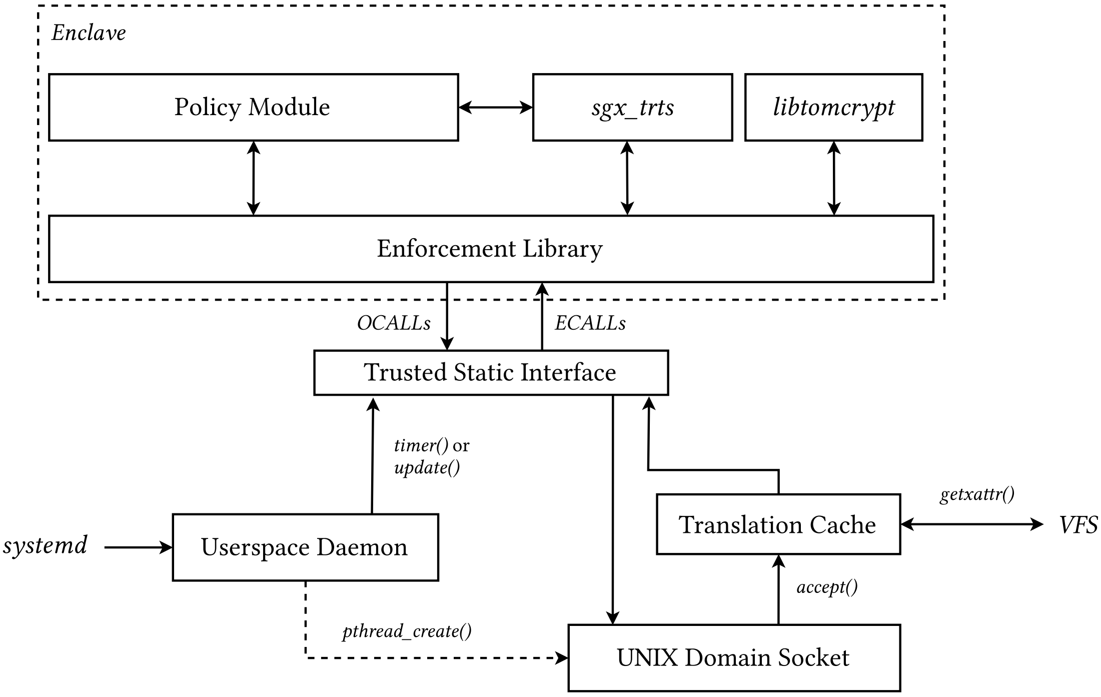
Enclave
Policy Module
sgx_trts
libtomcrypt
Enforcement Library
OCALLs
ECALLs
Trusted Static Interface
timer() or
update()
systemd
Userspace Daemon
Translation Cache
getxattr()
VFS
accept()
pthread_create()
UNIX Domain Socket

Figure 4.1: Abstract overview of an SGX enclave's protections.

Chapter 5

Evaluation

Chapter 6

Summary and Conclusions

Bibliography