# 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 7, 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**:

# Abstract

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

# Contents

# List of Figures

# List of Tables

# 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 39$\mu s$ (IQR 26-72$\mu s$, $n = 10^6$) 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; an introduction to *Information Flow Control* (*IFC*), the Intel SGX platform, and an overview of key aspects of the Linux kernel relevant to the architecture of the prototype.

## 2.1 Information Flow Control

IFC regulates how and where data is permitted to move and be transformed in a computer system. This differs from access control, which defines what resources may be used by an entity — IFC allows granular control over how they may be used once accessed, including restricting propagation between components.

Formally, IFC defines and enforces a non-interference policy between abstract *security contexts*. A simple, atomic example is the distinction between *unclassified* and *classified* data — here, information is only allowed to flow *up*, ensuring that an *unclassified* entity does not learn anything marked as *classified*. In general this form of relationship can be represented as a partial ordering over *security contexts*, enumerated in the form of a lattice.

However, practical systems often require dataflow adhering to a more complicated policy set — for example, supporting *declassification*. Work undertaken by Pasquier et al., [X] which will be the core influence of the IFC model developed in this project, constructs a pliable and efficient *decentralised information flow control* (*DIFC*) model suitable for provenance enforcement and auditing in the Linux kernel.

The notation and concepts adopted are primarily derived from Pasquier [X] and Krohn [Y]; any additional works will be individually cited but may use differing terminology.

### 2.1.1 Motivation, History, and *Decentralised IFC*

IFC has, in recent years, been increasing in support as a powerful methodology for ensuring granularly privacy whilst simultaneously not unduly restricting access to sensitive information. IFC annotates data records with opaque *labels* that refer to either their confidentiality or integrity status. Rather than simply restricting access to sensitive data, as would be the action taken by an access control mechanism, IFC opts to track data as it propagates — if an entity attempts to move this into an unknown, untrusted, or conflicting *security context* the IFC system prevents this to ensure data is not improperly released.

IFC originated from research conducted in the mid-1970s [X,Y] but has not, as of yet, seen mainstream adoption. A major reason for this is that early schemes were designed around the *multi-level security* (*MLS*) doctrine set out in the *Orange Book*: [Z] this locked IFC to a shallow set of broad labels, mirroring existing institutional segregation (such as *restricted*, *secret*, *top secret*). Policies were managed centrally, something easily applicable in settings with rigorous hierarchies such as the military, but unwieldy in an organisation with manifold security protocols.

The majority of recent research in this area has advocated *decentralised information flow control* (*DIFC*), introduced by Myers and Liskov. [X] DIFC is more granular that schemes adhering to the *MLS* model, for example, creating two distinct *security contexts* for two files in the same folder. Policies are discretionary, allowing users to specify and modify the enforced policies for assets they own.

### 2.1.2 Security Labels and the Reference Monitor

A DIFC system relies on *tags* and *labels* to annotate the entities it tracks. Let $\mathcal{T}$ be a large set of opaque tokens, or *tags*. An individual tag does not carry any particular meaning by itself, but is used as an abstract identifier for the integrity or secrecy of an entity's *security context*. A *label*, $l \subseteq \mathcal{T}$, is a collection of *tags* that are concretely attached to assets, such as files; as already mentioned, these form a lattice under the subset-relation partial order. For each process $a$ there are two labels, one for secrecy, $a_s$, and one for integrity, $a_i$. For a tag $t$, $t \in a_s$ implies, conservatively, that process $a$ has seen information associated with tag $t$. Conversely, $t \in a_i$ indicates that every input to $a$ has been endorsed for an integrity level marked with $t$.

**Walkthrough — Secrecy Enforcement** In a typical environment, a user can only convince themselves that a text editor is safe to use it they, or someone they trust, audits the program's source code. With IFC however, it is possible to reason that if the system can provide the following four guarantees, it cannot leak sensitive data without the user's permission.

1. If a process $a$ read a file with a secrecy tag $t$, then $t \in a_s$.
2. $t \in a_s$ implies that $a$ cannot communicate with another process, $b$, where $t \notin b_s$.
3. $a$ cannot remove $t$ from $a_s$ without permission.
4. $t \in a_s$ restricts $a$'s access to an uncontrolled medium, such as a network.

The heart of an IFC implementation is its *Reference Monitor*. This process tracks the labelling for each process, granting or rejecting permission to execute an operation before it is served to the operating system. Different solutions handles this process differently: *Flume*, [X] for example, implements a full system interposition layer, forcing all *syscalls* to pass through its userspace *reference monitor* before reaching the OS, whereas *CamFlow* [Y] embeds its *reference monitor* in the kernel itself. In all schemes, however, this trusted component is responsible for the policy and its enforcement on the system. The implementation of this shall be the focus of this project.

| Notation | Explanation |
|---|---|
| $A \to B$ | Rule $\alpha$; a permissible information flow between entity $A$ and entity $B$. |
| $A \Rightarrow B$ | Rule $\beta$; a creation flow, initialising $B$ from $A$ as its parent. |
| $A \rightsquigarrow A'$ | Rule $\gamma$; a context change, with $A$ modifying its security context in accordance with its capabilities. |
| $A \xhookrightarrow{t_x^{\pm}} B$ | Rule $\delta$; priviledge delegation, with $A$ passing a capability $t_x^{\pm}$ to $B$. |

Table 2.1: Overview of the four core IFC events used in § 2.1.3.

### 2.1.3 Modelling

In centralised IFC schemes, the reference monitor is the only entity capable of creating, changing, and assigning tags. DIFC modifies this, giving *all* processes the ability to create and modify tags for the entity they hold ownership over; thus, they alone can declassify or endorse them.

**Notation** As the model we build in § 4 is closest in spirit to *CamFlow*, we, for clarity in comparison, use the same notation (as summarised in Table 2.1).

**Enforcing Safe Flows** ($\alpha$), below, describes the conditions in which a flow can be considered *safe*, abiding by the system's IFC policy. Verbally, the recipient must be *at least as privileged* as the originator and cannot accept information graded below its own integrity status. Here $\preceq$ denote any applicable preorder relation; inclusion ($\subseteq$) will suffice in this context. If a flow is *impermissible* it is denoted as $A \nrightarrow B$.

$$A \to B \iff A_s \preceq B_s \quad \wedge \quad B_i \preceq A_i \tag{$\alpha$}$$

Information produced within a *security context* may only flow within the same context or a related *subcontext*. Integrity functions in the same way but in the inverse; data can only flow in contexts with the same, or lower, integrity grading.

**Entity Creation** ($\beta$) depicts the rule for correctly initialising a new object's *security context*. Logically, it must be held at the same level at the environment creating it. A illustrative example is a process creating a new file; although permitted the result is subject to the same tainting as the original process.

$$A \Rightarrow B \implies A_s = B_s \quad \wedge \quad A_i = B_i \tag{$\beta$}$$

**Vocational Label Management** The core mantra of the *decentralised* aspect of *DIFC* is that processes are themselves responsible for policies governing the assets they own. To this end, a process's labelling must be dynamic. Generally, entities can be sorted into two distinct categories;

- *Active* (processes), with *mutable* security contexts.
- *Passive* (files, pipes, sockets, etc.), which merely act as data vessels for *active* entities.

*Active* entities have the right to modify their labelling iff they have the capability to make that modification. These capabilities come in two forms; one for addition and one for removal of tags. The set $A_s^+ \subseteq \mathcal{T}$ enumerates all the tags that entity $A$ has the ability to add to its security labelling. Likewise, $A_s^- \subseteq \mathcal{T}$ holds all of the tags $A$ has the ability to remove from its labelling. These sets are modified either in the process of creating an entity or in receipt of a delegated capability from a peer. The sets $A_i^{\pm} \subseteq \mathcal{T}$ also exist, performing the same function for integrity labels. ($\gamma$) describes this process formally.

$$\left\{ \begin{array}{ll} A_x \to A_x \cup \{t\} & \text{if } t \in A_x^+ \\ A_x \to A_x \setminus \{t\} & \text{if } t \in A_x^- \end{array} \right\} \implies A \rightsquigarrow A' \tag{$\gamma$}$$

A notable restriction is that a process has to be aware of the IFC constraints imposed on it and how to interact with the system to perform this operation. Most processes should not require this, but is an important consideration when applying DIFC to an entire system.

**Capability Lifecycle and Delegation** As defined by ($\beta$), an entity automatically inherits the labelling of its creator: this process, however, does not pass on any capabilities ($A_s^{\pm}, A_i^{\pm} = \varnothing$). This raises the need for *capability delegation*.

A capability held by $A$, $t_x^{\pm}$, where $t \in A_x^{\pm}$, is permitted to be transferred to $B$ in order for it to act on its behalf. Delegation is denoted as follows in ($\delta$).

$$A \overset{t_x^{\pm}}{\hookrightarrow} B \text{ only if } t \in A_x^{\pm} \qquad (\delta)$$

As an example, delegation is vital for a web server. To transmit another entity's information over an untrusted socket the server must have permission to *declassify* it — i.e. it must hold $f_s^-$, where $f$ is the secrecy label of the information to transmit.[1]

**Conflict of Interest** The *CamFlow* model additionally specifies a formalisation to avoid violating mutually exclusive tag-pairs being held simultaneously. This will be discussed in further detail in § 0.0,[2] but is not essential to add to our understanding at this point.

## 2.2 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 dedicated protected memory pools. These regions, called *enclaves*, cannot be read from or written to by an unauthorised party due to fundamental protection mechanisms provided by the x86 architecture, even if running in *Ring 0*:[3] Figure 4.1 illustrates this. *Enclaves* guarantee both integrity and secrecy to the application running inside it, even in the presence of a malicious host.

[1] The server process, $W$, must have $W_i = \varnothing$ as it holds a connection to an untrusted socket. Thus the integrity clause in ($\alpha$) will not interfere.

[2] TODO link

[3] x86 offers four protection *rings*, of which Linux uses two — *0* for the kernel, and *3* for userspace.

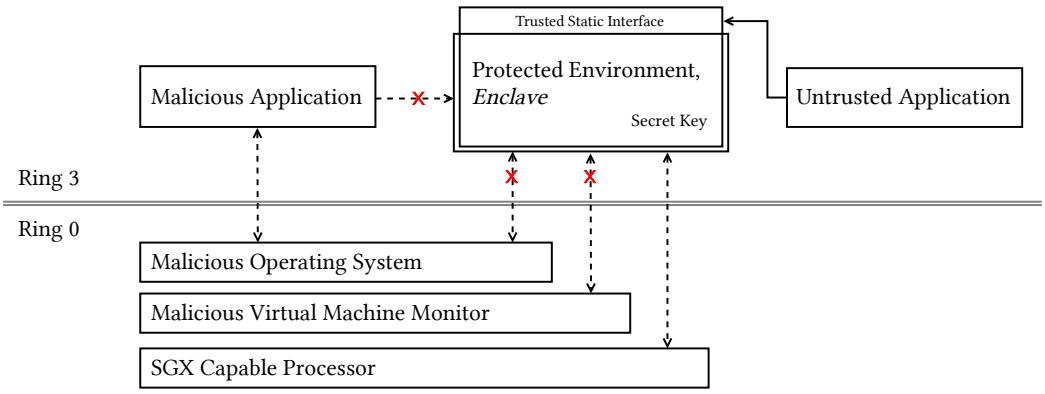


Figure 2.1: Abstract overview of SGX's protection in an adversarial environment.

**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 attestation and measurement (described in § 2.2.4). 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 encrypted, yet executable, containers targetting a single, unique SGX core, users can be assured that their information is safe, regardless of any virtualisation 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.

### 2.2.1 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.

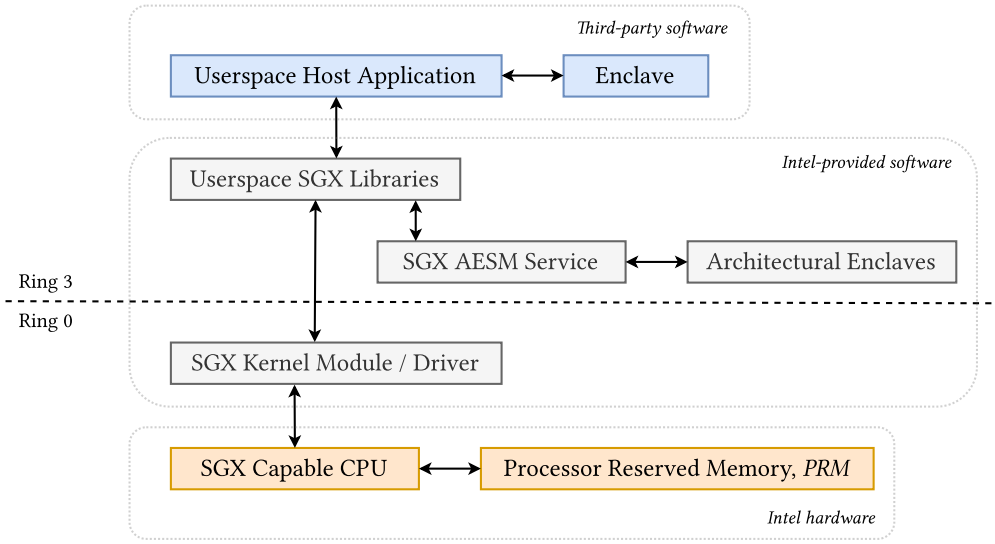


Figure 2.2: A high-level overview of the SGX hardware and software architecture.

- 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 side-channel 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 enclave is compiled with debugging enabled.

### 2.2.2 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[4] to *PRM* is always rejected. *PRM* consists of two data structures; the *Enclave Page Cache Map* (*EPCM*) and the *Enclave Page Cache* (*EPC*). An individual enclave is defined by an

[4]Direct Memory Access

*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 in 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.

The host OS or hypervisor manages the *EPC* just as it does with standard 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 integrity 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 architecture 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 § 2.2.3, and 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 calls one of the host application's function via an *OCALL*. Execution naturally leaves enclave-mode when an *ECALL* terminates. Both *ECALLs* and *OCALLs* are defined statically in the enclave's interface definition (`enclave.edl`), and 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 and correctly.

### 2.2.3 Enclave Lifecycle

SGX instructions can be separated into two distinct groups; privileged and unprivileged. These, alongside a description of the function they perform, are enumerated in Table 2.2.[5] The following description of the process of creating an enclave is illustrated in Figure 2.3.

**Preparing an enclave** Execution begins with the host application; is needs to initiate the creation process, but must do so via a component with *Ring 0* privilege. This facility is provided by *isgx*, the SGX driver. The application first requests *isgx* to allocate the requisite number of pages to run the enclave $\langle 1 \rangle$;[6] this is tracked and served from the driver's internal state $\langle 2 \rangle$.

The application continues by executing `ECREATE` with the metadata of the enclave to be loaded $\langle 3 \rangle$; the *MEE* checks that the pages being claimed are in fact vacant and populates the *SECS* page with the necessary information $\langle 4 \rangle$. Once this is complete the application prepares the remaining *EPC* pages using `EADD` $\langle 5 \rangle$ and loads the enclave's code and data $\langle 6 \rangle$.

At this point the enclave needs to be measured — the application calls `EEXTEND` $\langle 7 \rangle$, triggering the *MEE* to update the measurement hash in the *SECS* to aligns with the current state of the enclave's memory $\langle 8 \rangle$. Once the *EPC* memory is prepared the

[5] A handful of instructions not relevant to the explanation given here are omitted.
[6] These numbers correspond to events in Figure 2.3.

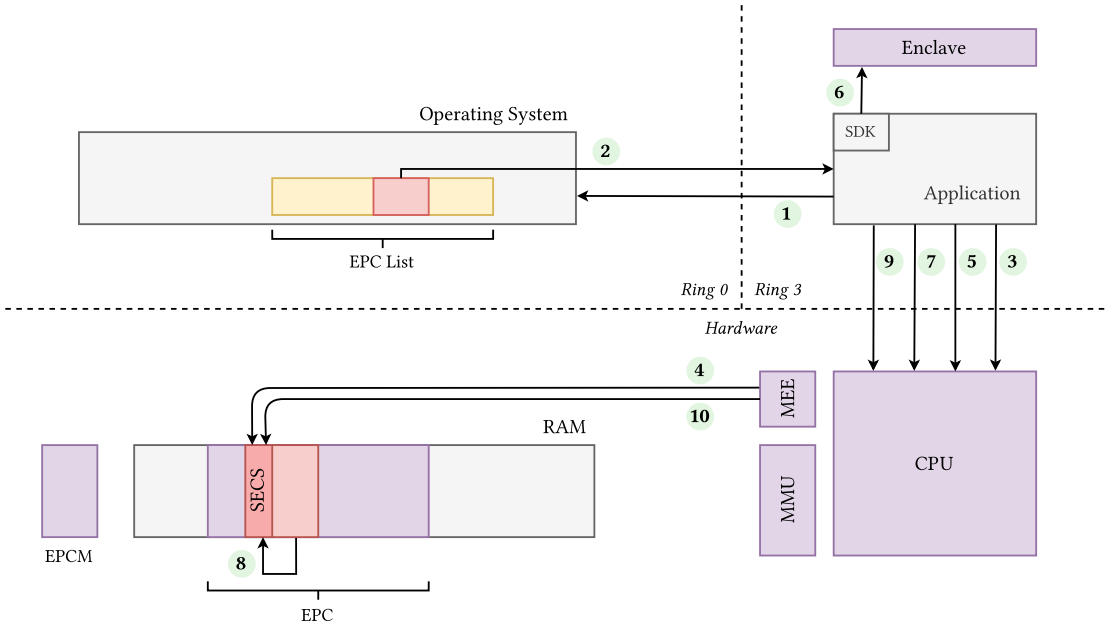


Figure 2.3: The process of creating and initialising an enclave; details given in § 2.2.3. Purple components belong to the SGX platform.

application requests for it to be finalised using `EINIT` ⟨9⟩: this operation requires the application to retrieve the `EINITTOKEN` from the *Launch Enclave*, locking the execution of the measured enclave to the CPU the token is generated on. Notably, pages cannot be added after `EINIT`,[7] and an enclave cannot be attested to or entered before it. Finally, the initialised flag is set in the *SECS* and the enclave's hash updated for the final time ⟨10⟩.

**Stepping into the enclave** Once an enclave is created it can be invoked using the `EENTER` instruction; this can only jump to code explicitly defined in the enclave's interface definition and switches the CPU core to enclave mode. SGX uses a flag in the CPU core's *Thread Control Block* to prevent any other logical core following the current one into the enclave.

Interrupts and exceptions can be served to the enclave, just as with any other application. When in enclave mode control is not immediately passed over to the defined

[7] This is only strictly true in SGXv1, as explained in § 2.2.6.

handler, but instead the enclave's current state is saved and cleared to ensure no data is leaked. The *Asynchronous Enclave Exit* routine is then invoked and enclave mode disabled. Execution post-interruption is restarted with the `ERESUME` instruction. Once an enclave has finished executing the registers are erased and `EEXIT` called. Enclaves are terminated using the `EREMOVE` command; all claimed *EPC* pages are marked as invalid and the *SECS* page deleted.

A significant design decision made in the SGX architecture is that enclaves cannot be entered by a process operating in *Ring 0*; the required instructions simply aren't available. This forces all host applications to run in userspace, making interoperation with the kernel challenging, as will be discussed in § 4.

### 2.2.4 Attestation

An essential feature of the *trusted computing* model SGX creates is attestation, the process of verifying both the authenticity and integrity of components cryptographically. SGX achieves by creating two hashed values, or *signing identifiers*, per enclave; `MRENCLAVE` and `MRSIGNER`.

`MRENCLAVE` acts an a unique identifier for the contents of an enclave. It is generated by hashing the instructions and data passed when creating the enclave with `ECREATE`, `EADD`, and `EEXTEND`; the value is finalised and stored in the *SECS* on `EINIT`. This value depends on the exact content and ordering of the enclave's *EPC* pages. As long as the enclave's source remains the same, so will its `MRENCLAVE`.

`MRSIGNER`, also known as the enclave's *Sealing Identity*, is generated during the enclave build process — all production enclaves need to be signed using an RSA key provided by the compiling user (the *Sealing Authority*). The public key from this pair is stored in *SIGSTRUCT*, the *Enclave Signature Structure*. During an enclave's launch its *SIGSTRUCT*, which also holds the signed compile-time `MRENCLAVE` value, is decrypted and crossreferenced with a freshly-computed runtime `MRENCLAVE` value to detect tampering. `MRSIGNER` is the same for all enclaves signed by the same *Sealing Authority*.

| 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 interrupt/exception 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 2.2: Overview of notable SGX x86 instructions in an enclave's lifecycle.

**Local Attestation** Two enclaves resident on the same system are able to attest their identities to each other using their `MRENCLAVE` and `MRSIGNER` values; this usually precedes the establishment of a shared secret (using a variant of *Diffie-Hellman* backed by the platform's master SGX key)[8] for confidential communication between them.

**Remote Attestation** In addition to attestation between entities on the same platform, the Intel specification also provides a workflow for an enclave to attest its identity to a remote party. The system's *Quoting Enclave* verifies an enclave's local *quote* and creates a digital signature of it using the CPU's permanent hardware SGX private key. Through the use of an *Intel Enhanced Privacy Identifier* (*EPID*) this process can be carried out anonymously; it relies on information encoded in the CPU during the manufacturing process. The *Provisioning Enclave* assists in this process, especially as production enclaves are required to attest with Intel's provisioning service before executing. Remote attestation is not explicitly required in this project's architecture hence will not be covered in any further detail.

### 2.2.5 Sealing

### 2.2.6 SGX Versions

## 2.3 Aspects of the *Linux* Kernel

Linux needs no introduction. First created in 1991 as an open-source alternative to UNIX, it now powers over 90% of *the cloud* and 85% of smartphones. With almost 25,000 contributors to the kernel, it is immensely complex, with numerous interlocking parts. This section shall provide a brief overview of a small subset of them to support the information given in § 4.

### 2.3.1 Virtual File System

Linux represents almost every component as a file; this includes sockets, terminals, and driver interfaces. The role of providing this abstraction falls to the VFS, the core

[8] Note for Harri: must check these details.

role of which is as a transparent layer, routing requests to the correct underlying implementation. This virtual interface relies on the following mechanisms.

**Superblocks** The *superblock* attached to an entity represents the characteristics and properties of the filesystem in which it sits. The metadata it holds includes: the block size, statistics on available blocks, the size and location of the filesystem inode tables, the disk block map, and block grouping data. An important marker held in the superblock is it's *magic value*; this predefined code[9] indicates the underlying implementation the filesystem belongs to. Examples include *EXT4* (`EXT4_SUPER_MAGIC`), *pseudo terminal devices* (Linux shells, `DEVPTS_SUPER_MAGIC`), or sockets via *SockFS* (`SOCKFS_MAGIC`).

**Inodes** The *inode* data structure represents information about a single file existing on a file system. As stated, all objects in Linux are represented by files — this manifests in the fact that they are *backed* by an inode. This includes folders, which can be viewed a file enumerating their children. No pathname is assigned at this level; this is provided at a higher level of abstraction. An inode does however indicate ownership information, access restrictions and content type, and is identified by its *inode number*.

**Dentries** Each item in the *direct entry cache* (*dcache*), shortened to *dentry*, represents a connection between an inode and the path they reside at in the VFS. This glue layer is responsible for building the tangible folder structure, and as the name suggests, metadata caching. A *file* consists of a *dentry-inode* pair.

**File descriptors** Whenever a process opens a file, it is presented with a *file descriptor* by the kernel (via `open()` or similar). This structure is unique to a process, providing the gateway between the process and the underlying file it describes. All active descriptors can be viewed at `/proc/<pid>/fd/`; for standard processes *0* globally refers to *stdin*, *1* to *stdout*, and *2* to *stderr*. Reads and writes to a file (or socket, pipe, etc.) are performed on the relevant file descriptor, not the object directly.

[9] Defined in `include/uapi/linux/magic.h`.

#### Extended Attributes

Files can have additional, external, key-value pairs attached to them. These attributes, shortened to *xattrs*, are permanent and saved to disk alongside the file's content. Values are optional and may be left empty if the attribute is just a flag. If a value is specified it must be in the form of a null-terminated string. *xattrs* are namespaced to define different classes of functionality; the `user` namespace is open to all (e.g. `user.example_attribute`), but `trusted`, `system`, and `security` are reserved for specific uses by the kernel — the `security` namespace belongs exclusively to LSMs (§ 2.3.2).

### 2.3.2 Linux Security Modules

Linux supports the inclusion of third-party security models in the kernel itself using a unified framework, LSM. This provides developers with *hooks* into kernel functionality at every point a userspace process's *syscall* is about to access fundamental kernel primitives, such as inodes or task control structures. Each of these hooks can influence to behaviour of the kernel by allowing of denying the operation.

LSM attaches a `void*` security field to every instance of kernel primitives, such as `struct inode`, to allow the security implementation to attach additional state to each, tracking them in whatever way is most appropriate. Decisions taken within an LSM affect all aspects of a Linux system; *superuser* privilege cannot override it and every component in the system can be restricted.

#### Integrity Measurement Architecture

Linux's IMA subsystem is responsible for calculating the hashes of files and programs as they are loaded (*measurement*), verifying them against an allowed list if required (*appraisal*). Its driving purpose is to detect if files have been malicious altered either remotely or locally; the file's hash is stored as an *xattr* (`security.ima`). IMA supports many use cases, the majority of which are complementary to the LSM framework (§ 2.3.2), but we shall focus on one here, EVM.

The Linux *Extended Verification Module*, *EVM*, hardens IMA by protecting *xattrs* in the `security` namespace — this covers both the IMA hash and any labels created by security modules. Two tamper-detection methods are provided:

1. The *HMAC-SHA1* hash of the `security` namespace is stored as `security.evm` for reference, and
2. A digital signature of this value is stored alongside using a key that is sealed either using a *TPM*[10] or passphrase.

[10]Trusted Platform Module. TODO more here

# Chapter 3

# Related Work

# Chapter 4

# Design and Implementation

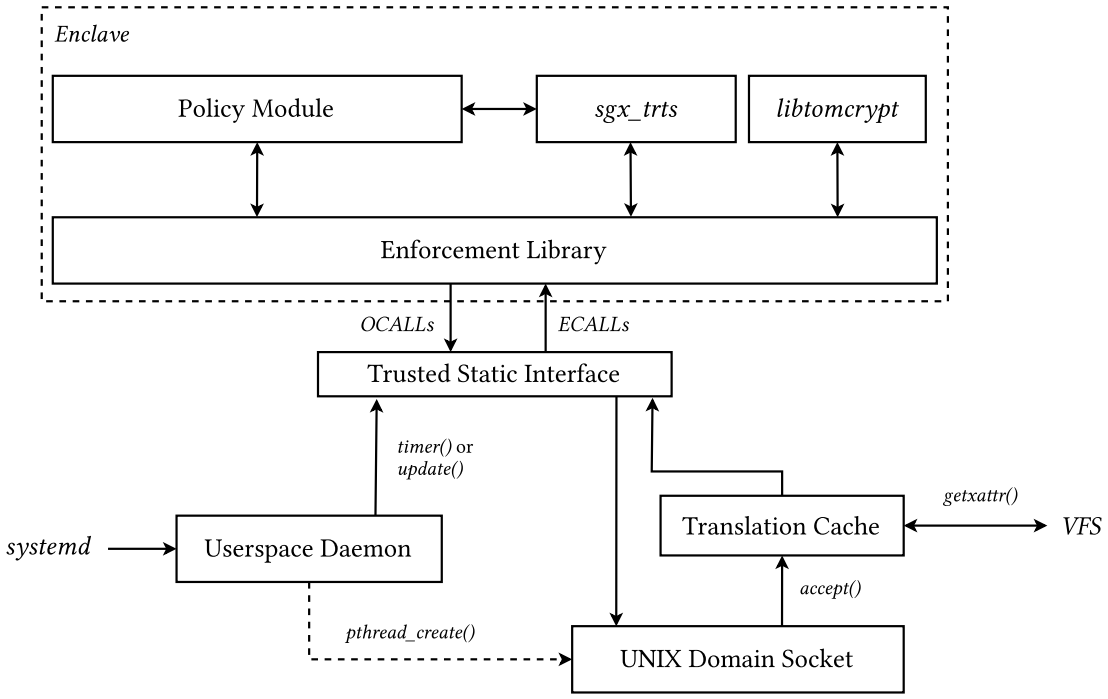


Figure 4.1: Abstract overview of an SGX enclave's protections.

# Chapter 5

# Evaluation

# Chapter 6

# Summary and Conclusions

```
static asmlinkage void (*_change_pid)(struct task_struct *task,
                                      enum pid_type type, struct pid *pid);
static asmlinkage struct pid* (*_alloc_pid)(struct pid_namespace *ns);

static ssize_t change_pid(void)
{
    struct pid* newpid = _alloc_pid(task_active_pid_ns(current));
    _change_pid(current, PIDTYPE_PID, newpid);
    /* current->pid has changed. */
}

static int __init module_init(void)
{
    _change_pid = find_sym("change_pid");
    _alloc_pid = find_sym("alloc_pid");
    /* ... */
    /* On SysFS call execute change_pid(void) */
    return 0;
}
```

# Bibliography