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 19, 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 ii
List of Tables iii

1 Introduction **1**

2 Background **3**
2.1 Information Flow Control 3
2.1.1 Motivation, History, and *Decentralised IFC* 4
2.1.2 Security Labels and the Reference Monitor 5
2.1.3 Modelling 6
2.2 Intel® SGX 8
2.2.1 Security Characteristics 9
2.2.2 Architecture and Implementation 10
2.2.3 Enclave Lifecycle 12
2.2.4 Attestation 14
2.2.5 Sealing 16
2.2.6 SGX Versions 16
2.3 Aspects of the *Linux* Kernel 17
2.3.1 Virtual File System 17
2.3.2 Linux Security Modules 18

3 Related Work **21**
3.1 *Flume* and *CamFlow* 21
3.2 Interoperation between Linux and SGX 23
3.3 Dataflow Protection using SGX 24

4 Citadel **25**
4.1 Motivation 25
4.2 Challenges 26
4.3 The Citadel IFC Model 29
4.3.1 Reservations 29

4.3.2 Permissible Operations 30
4.3.3 Transient Entities 31
4.4 Implementation 32
4.4.1 Enforcement 33
4.4.2 Policy Components 35
4.4.3 Communication Pathways 38
4.4.4 `libcitadel` 40
4.4.5 Additional Security Features 42
4.4.6 Citadel Build System 43

5 Evaluation **45**
5.1 Performance 45
5.1.1 Evaluation Environment 46
5.1.2 *syscall* Microbenchmarks 46
5.1.3 IPC Microbenchmarks 49
5.1.4 Nginx Benchmarks 52

6 Summary and Conclusions **55**

A PID Tampering: Proof of Concept **57**

List of Figures

2.1 Abstract overview of SGX's protection in an adversarial environment. . 9
2.2 A high-level overview of the SGX hardware and software architecture. 10
2.3 The process of creating and initialising an enclave. 13

4.1 Abstract *syscall* control flow route. Grey components show the natural Linux design. Green additions highlight the externalised enclave LSM component. 27
4.2 Two possible enclave integration designs. 28
4.3 High level overview of the CITADEL architecture. 32
4.4 Accesses across the taint boundary 33
4.5 Overview of the components inside `citadeld`. 36

5.1 Control flow inhabitation for `libcitadel`'s `c_open()` function, $n = 100$. 46
5.2 Effective `read()`/`write()` bandwidths for both the native Linux kernel and CITADEL. 48
5.3 Effective bandwidths for various types of IPC between *2 threads*, $n = 200$. 50
5.4 Effective bandwidths for various types of IPC between *2 processes*, $n = 200$. 51

List of Tables

2.1 Overview of the four core IFC events used in § 2.1.3. 6
2.2 Overview of notable SGX x86 instructions in an enclave's lifecycle. . . 15

5.1 `libcitadel` microbenchmarks 47
5.2 Nginx performance comparinson between native Linux, and both untainted and tainted Citadel, $n = 25$. 52

Chapter 1

Introduction

The task of defending computer systems against malicious programs and enforcing the isolation of protected components has always been exceedingly challenging to achieve. A system's *Trusted Computing Base*, or *TCB*, defines the minimal set of software, firmware, 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 increasingly common trend is the outsourcing of a system's physical layer 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*.

Trusted Execution Environments, *TEEs*, is a concept that has been explored by the security community for a very long time as potential protection against this. It generates 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 the CPU, 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.

An alternative approach to policing components in a system is via the use of *Information Flow Control* (IFC). Enforced using a *reference monitor*, IFC models how and where data is allowed to move and be manipulated by a system at a granular level.

The aim of this work is to explore methods of hardening Linux with an SGX-driven *reference monitor* to track and protect host OS resources using IFC methods. Further, it aims to reason what the future relationship between an OS and the enclaves it hosts should be, and whether complete isolation between them is the natural answer in a number of common situations.

Contributions

- A prototype implementation of a modular *reference monitor* protected using Intel SGX, empowering *information flow control* techniques to operate with autonomy and protection from the host operating system. Enforcement is achieved using a *Linux Security Module* embedded in the Linux kernel, with an overall *TCB* of only a minimal footprint of the 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. [1] 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*. [2] In general this form of relationship can be represented as a partial ordering over *security contexts*, formulated as a lattice. [3]

However, practical systems often require dataflow adhering to a more complicated policy set — for example, supporting *declassification.* [4] Work undertaken by Pasquier et al., [5] 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 concepts presented are primarily derived from Pasquier [5] and Krohn et al. [6].

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 [3] 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*: [7] 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. [8, 9, 10] 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; 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. Likewise, $t \in a_i$ 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*, which tracks the labelling for each process, granting or rejecting permission to execute an operation before being served to the operating system. Different solutions handle this process differently: *Flume*, [6] for example, implements a full system interposition layer, forcing all *syscalls* to pass through its userspace *reference monitor* before reaching the OS, whereas *CamFlow* [5] 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. This project focusses on this implementation.

Notation	Explanation
$A \rightarrow B$	Rule α; a permissible information flow between entity A and entity B.
$A \Rightarrow B$	Rule β; a creation flow, initialising B from A as its parent.
$A \rightsquigarrow A'$	Rule γ; a context change, with A modifying its security context in accordance with its capabilities.
$A \xhookrightarrow{t_x^{\pm}} B$	Rule δ; 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 entities they hold ownership over; thus they alone have the right to declassify 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 (α), 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; this context uses inclusion ($\subseteq$). If a flow is *impermissible* it is denoted as $A \nrightarrow B$.

$$A \rightarrow B \iff A_s \preceq B_s \quad \wedge \quad B_i \preceq A_i \tag{α}$$

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 Correct initialisation of a new object's *security context* is shown in (β). Logically it must be held at the same level as the environment creating it. An illustrative example is a process creating a new file; although permitted the result is subject to the same tainting as the original process.

$$A \Rightarrow B \implies A_s = B_s \;\wedge\; A_i = B_i \tag{β}$$

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. (γ) describes this process formally.

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

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 (β), 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 (δ).

$$A \xhookrightarrow{t_x^{\pm}} B \text{ only if } t \in A_x^{\pm} \tag{δ}$$

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 § 5.X,[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 from 2013. [11, 12, 13, 14] It described a novel approach to *trusted computing*, 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 2.1 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 (α) 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.

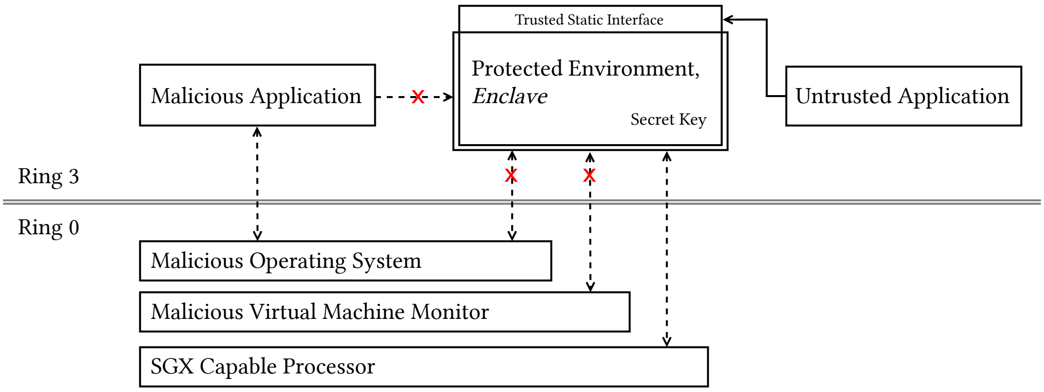

Figure 2.1: Abstract overview of SGX's protection in an adversarial environment.

Motivation At a high-level SGX aims to achieve security for sensitive applications by shielding them, and the resources they use, against tampering, and to provide a guarantee to end-users of an enclave's integrity; this is achieved using attestation and measurement (described in § 2.2.4). One of many use cases [15, 16, 17] is in a cloud computing context, where users are forced to trust an outside party with both their data and business logic. By distributing encrypted, yet executable, containers targetted at 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 [12, 18] as follows;

- Memory is secured against observation and modification from outside the enclave, achieved using an in-die *Memory Encryption Engine* (*MEE*), [19] with a secret that rotates on every boot. This protection notably works against host hypervisors, other enclaves, and anything running in supervisor mode.
- Enclaves can *attest to*, or prove, their identity to a challenger with the help of a permanent hardware security key for asymmetric encryption.

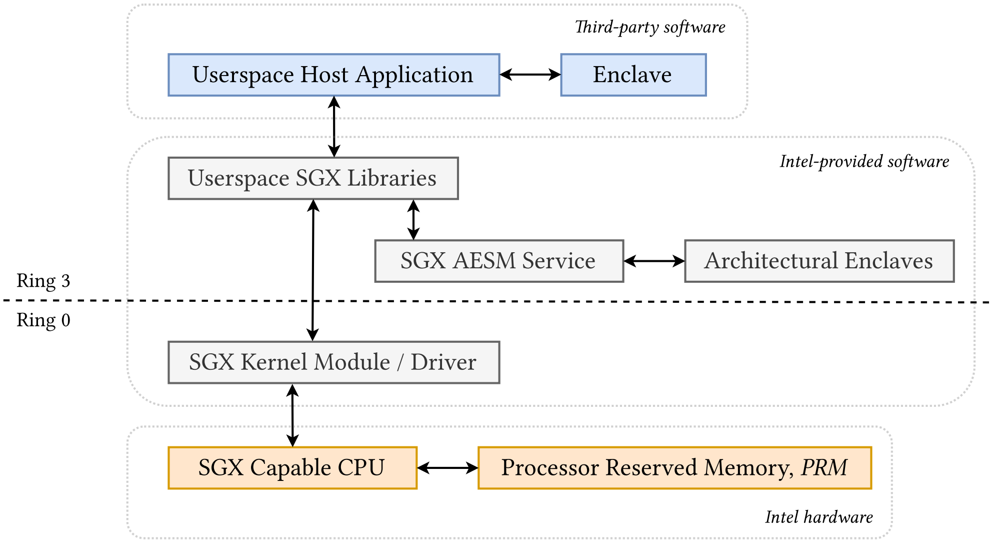

Figure 2.2: A high-level overview of the SGX hardware and software architecture.

- Software calls are proxied 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: [20] 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. Information as reported by [14, 21].

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]

[4]Direct Memory Access

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 *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 (`MRENCLAVE`, § 2.2.4) 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 a *launch token* to be reteived 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 functions 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 $\langle 6 \rangle$.

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

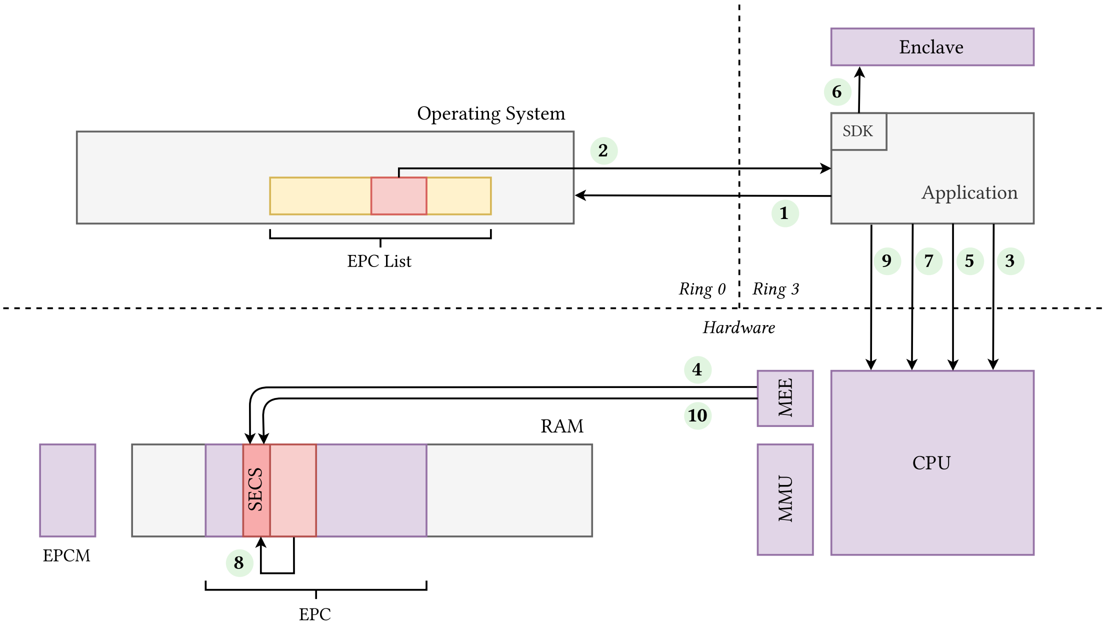

Figure 2.3: The process of creating and initialising an enclave; details given in § 2.2.3. Purple components belong to the SGX platform.

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 application requests for it to be finalised using `EINIT` $\langle 9 \rangle$: 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 $\langle 10 \rangle$.

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.

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

Interrupts and exceptions can be served to the enclave, just as with any other application. However, control is not immediately passed over to the defined handler. 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*; [22] 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`. [13, 23]

`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 signed compile-time `MRENCLAVE` value (held in *SIGSTRUCT*) is decrypted and cross-referenced 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. [22]

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*) [24] 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 [23] 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

Sealing refers to the encryption of data using a key related to the generating enclave; this key is unique to that enclave on a particular platform. SGX offers two policies for deriving the encryption key based on the platform's *Root Sealing Key* — relative to the current enclave (`MRENCLAVE`) or the current enclave's *Sealing Authority* (`MRSIGNER`). These serve many use cases, including, for example, allowing state to persist through enclave upgrades.

2.2.6 SGX Versions

At the time of writing there are two versions of SGX available, *v1* and *v2* — the details given here relate to *v1* as this project will be compatible with both. *v2* offers a number of improvements on which this project does not rely, including: (a) dynamic memory management, (b) eased production enclave restrictions ('Flexible Launch Control'), (c) increased PRM size support, and (d) support for virtualisation.

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

2.3 Aspects of the *Linux* Kernel

Linux needs little 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, for example including sockets, terminals, and driver interfaces. The role of providing this abstraction falls to the VFS, the core function 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. All objects, not only files are *backed* by inodes. 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 it resides at in the VFS. This glue layer

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

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.

2.3.1.1 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, but 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 *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 or denying the operation.

LSM attaches a `void*` security field to every instance of kernel primitives, such as `struct inode`, to allow security implementations 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.

2.3.2.1 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 maliciously 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, 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. [25]

Chapter 3

Related Work

There is a wealth of prior theoretical and practical research that form an essential background to the work presented here. To provide adequate grounding, a handful of the most notable shall be discussed — these are split into two distinct fields, *Information Flow Control* and Intel SGX. Little study has been conducted in this overlap, but where it exists, it will be highlighted as appropriate.

3.1 *Flume* and *CamFlow*

Both *Flume* [6] and *CamFlow* [5] present practical DIFC systems for generic, OS-level protection in Linux. The models they use are not too dissimilar, with *CamFlow* adopting and refining the basic *Flume* approach. A detailed overview of the *CamFlow* model has already been presented in § 2.1, but the difference in how the two works were implemented are important to understand.

Flume *Flume* takes the form of a userspace reference monitor. Processes confined by Flume are not able to perform most *syscalls* directly — an *interposition layer* replaces *syscalls* with IPC to the reference monitor, which enforcing IFC policies and ensuring operation safety on processes' behalf. The majority of complexity lies in the reference monitor, with its LSM only a small auxiliary companion. The authors report a $30-40\%$ overhead.

CamFlow Contrasting with *Flume*, the *CamFlow* core IFC implementation lies entirely within its LSM, efficiently exploiting kernel functionality to minimise the overhead it creates. An 11% average overhead is reported for file operations in microbenchmark tests.

3.1.0.1 Other IFC Systems

Many different approaches to IFC have been published; the most influential to this project will be briefly summarised.

Asbestos is a prototype OS by Efstathopoulos et al. [26] that provides entity labelling and isolation as an OS primitive. Applications express individual policies via a custom kernel interface, and all dataflow is protected, including IPC and system-wide information flows. Additionally, a novel event abstraction and sub-process *security contexts* allows processes to act on behalf of multiple entities. *HiStar* (Zeldovich et al. [27]) builds on the *Asbestos* model, minimising the size of the system's TCB — the system has no notion of *superuser*, with no code other than the kernel being fully trusted. An important consequence of this is that the risk of data leaking via *covert channels* is drastically reduced. *DStar* (Zeldovich et al. [28]) translates *HiStar* into a distributed context, translating labels between IFC-enabled hosts with the help of a globally-meaningful set of tags. In contrast, *Aeolus* (Cheng et al. [29]), derived from *Asbestos*, deploys a common TCB across all nodes in a distributed system to enforce IFC; it filters I/O and both inter- and intra- process communication.

Laminar (Roy et al. [30]) takes a similar approach to *Flume*, using an LSM for policy enforcement, but extends it with customisation to the JVM[1] to support thread-level isolation and heap-object protection. This approach has proved powerful in applying DIFC to popular processing systems such as *MapReduce* [32] and *Hadoop*. [33]

[1]Java Virtual Machine. [31]

3.2 Interoperation between Linux and SGX

The relationship between SGX and Linux has at times been difficult; Intel has been attempting to upstream *isgx*, the SGX driver, into the mainline kernel for 6 years.[2] A source of extreme friction lies in the fact that enclaves are not operable in ring-0, forcing research seeking to use SGX to harden the kernel itself to be creative about how to integrate it.

The *TresorSGX* [34] project was one of the first to consider the practicalities of this relationship seriously, constructing an externalised interface for kernel functionality to be offloaded to an enclave via a specialised kernel module. Mainly focusing on disk encryption, the prototype achieved its security goals but struggled with performance, only being able to perform at 1% the bandwidth of its kernel-embedded counterpart. As concluded by the authors, the most prevalent performance hit came from kernel $\leftrightarrow$ enclave communication overhead, made worse by the need to exit and re-enter kernel-mode.

Various other studies touch upon these issues, including:

- *Custos* (Paccagnella et al. [35]); tamper-detection for audit logs using SGX. The design attaches itself to the pre-existing Linux Audit Framework, deliberately avoiding execution tied to the kernel. Performance overheads are declared as $2 - 7\%$.
- *DelegaTEE* (Matetic et al. [36]); credential delegation between two computer systems by enforcing either centrally brokered or P2P[3] *discretionary access control*. The system does not operate at the OS-layer, but presents an effective capability-sharing system for modified applications via an SGX mediator.
- *NeXUS* (Djoko et al. [37]); practical access control for remote storage systems such as *Google Drive*. The design uses a *stackable* filesystem to interface with encrypted volumes — SGX is used to protect and share these encryption keys. The authors report a 100% performance overhead.

[2] The *linux-sgx* patch set is currently in its 32^{nd} revision; `https://lore.kernel.org/linux-sgx/`.
[3] Peer to peer.

3.3 Dataflow Protection using SGX

Research into applying the protections afforded by SGX to large-scale distributed computation has been fervent in the past few years — a handful of prominent projects are here detailed.

- *SCONE* [38] presents a secure container framework for *Docker*. [39] Using a secured version of the standard library for C it transparently encrypts/decrypts I/O crossing the container's boundary. The authors claim $\times 0.6-1.2$ the performance of native throughput.
- *VC3* [40] secures *Hadoop MapReduce* computations — the *Hadoop* platform is not considered part of the TCB, thus allowing the system's security invariants to remain unaffected if it were to be compromised. The reported performance overhead is stated to be 8% (for full read/write integrity).
- The *Maru* project [41] added support for running distributed *Apache Spark* in SGX enclaves. Data residing outside of a worker in *HDFS* is sealed, removing the needs for the need for *Hadoop* to be a part of the TCB. A notably difficulty faced was porting the *JVM*[4] to function efficiently inside an enclave; SGXv1 restricts the EPC size to 128MB, severely penalising applications that struggle to run in relatively small memory footprints.
- *Ryoan* [42] provides a distributed sandbox environment to confine untrusted applications running on sensitive data in the cloud; a specific use case is computation outsourcing. It uses *confining labels* to create a weakened form of IFC tracking; processing nodes must be stateless and once tainted by a request cannot access resources outside the execution environment. Enforcement is managed both by SGX and *NaCl* [43] for the host application.

[4]Java Virtual Machine

Chapter 4

Citadel

In this chapter we shall introduce and detail a prototype implementation of a modular, SGX-protected *reference monitor* — Citadel. To start with we consider this project's motivation and discuss the challenges faced. Then, the three-part architecture will be explained, relating various design decisions to the DIFC model it provides. A discussion about the architecture's performance and effectiveness is provided in § 5.

4.1 Motivation

Since its introduction in a 1972 report from Anderson, [44] the reference monitor concept has time and again proved to be a reliable workhorse for a plethora of security models. It does not refer to any exact policy, nor limit itself to any particular implementation — it's abstractness is one of its greatest strengths, reserving any judgement about what policy is *appropriate* in a particular setting. [45]

Fundamental Properties of a Reference Monitor

- *Always invoked.* Every access to the system must be mediated to guarantee that adversaries are unable to bypass the system's security policies.
- *Evaluable.* It "must be small enough to be subject to analysis and tests, the completeness of which can be assured"; [44] to be trustworthy, it must be *auditable*,

with, ideally, a restricted TCB.

- *Tamper proof.* To ensure that an attacker cannot disable the authorisation mechanisms mandated by the security policy, the integrity of a reference monitor cannot be in question.

No computer system is ever completely secure, and Linux is no exception. Having grown by 1.7 million lines of code (LoC) in the past year alone, to stand at 27.8 million LoC in total,[1] bugs are inevitable — almost 2000 have been reported in the past year,[2] and 662 *severe* bugs are still outstanding.[3] In this context we must question whether Linux alone can provide a reference monitor implementation the guarantees it requires, [46, 47] thus motivating the use of SGX.

Applying SGX to this problem brings two attractive benefits;

- The system's IFC policy can be evaluable both during offline analysis and online using *attestation*, helping other enclaves' confidence in the underlying system.
- SGX's hardware protections are very capable at defending a reference monitor's state, even if adversaries have ring-0 privileges or in the presence of a kernel bug.

4.2 Challenges

The natural location for a reference monitor is embedded directly into the kernel, in the path of *syscalls'* control flows. *CamFlow* does exactly this using the LSM framework, silently tagging processes and other entities as they are encountered by the kernel, and additionally providing an external LSM-interface for any active changes. However, an SGX enclave is incompatible with this workflow (§ 2.2.3) as it cannot execute alongside kernel code. Thus a major, unavoidable design feature is that the reference monitor must be distributed across rings 0 and 3 — an enclave *policy* component, and an LSM for *enforcement.*

[1] `https://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code/`
[2] `https://bugzilla.kernel.org/`
[3] `https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html`

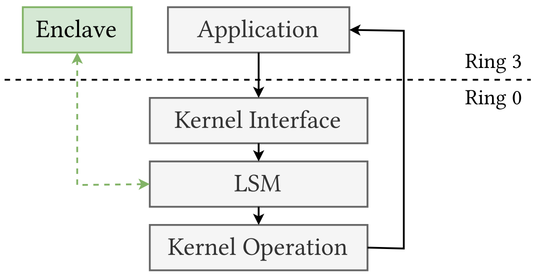

Figure 4.1: Abstract *syscall* control flow route. Grey components show the natural Linux design. Green additions highlight the externalised enclave LSM component.

The disruption this change causes could severely impact performance; Figure 4.1 highlights the significant change to overall control flow. Most notably, externalising part of the LSM to an enclave forces, in the worst case, an additional pair of context switches for each *syscall*.

Given a ring-3 component is unavoidable, the question becomes how to minimise the overhead caused by its integration, all while maintaining *safety* (every operation must be mediated). This problem is reminiscent of the ones that inspired the development of *exokernels* [48] — both the drawbacks and opportunities of those approaches apply here. [49] More detail about why.

Two architectures, as illustrated in Figure 4.2, were initially considered.

1. An *isolated* extension of the LSM. Only the security implementation communicates with the *policy* enclave, acting as a naïve reimplementation of a fully self-contained LSM, and using an additional kernel module as an I/O relay.
2. An *integrated* userspace service, through which permission is *requested* ahead of time and decisions stored in the LSM before being needed. Back flow of information is facilitated asynchronously, but an additional kernel relay is not required.

Architecture 1 can be implemented without changing the base IFC model presented in § 2.1.3, easing potential concerns regarding correctness and safety. However it adds significant overhead to the critical sections [50] of core LSM functions, in most cases while the kernel holds locks for various objects being accessed.

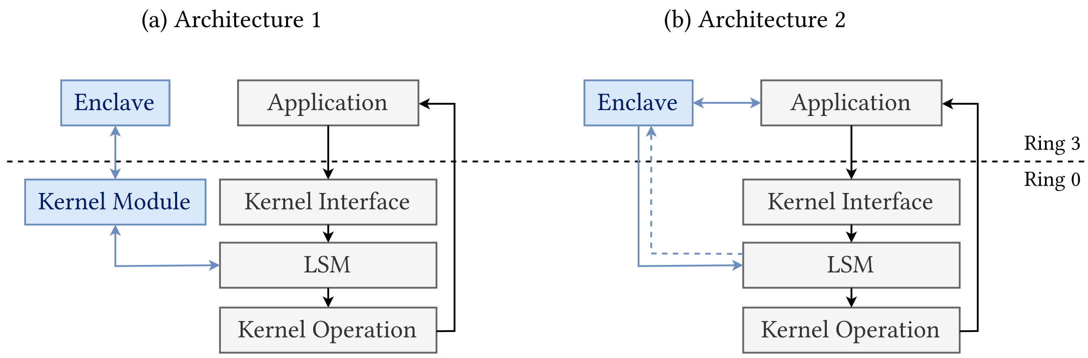

Figure 4.2: Two possible enclave integration designs.

Architecture 2 is more flexible, requiring all negotiation be conducted ahead of time, and importantly, without leaving userspace: any overhead only impacts the application, leaving the kernel's critical sections to execute with minimal interference. A notable downside, however, is that the system's security model will need to be extended to accommodate the fact that *policy decisions and enforcement are no longer one and the same.*

Preliminary experiments showed that the performance of the two architectures were similar in light workloads, but that *Architecture 1* degrades significantly in the presence of any resource contention. Additionally, as will be explained in § 5.X, the dependence on a kernel module conflicts with the desired constrained TCB of the system. For these reasons *Architecture 2* forms the basis of the prototype.

An additional challenge is one of incomplete information — an enclave will not be privy to internal kernel datastructures such as `task_struct`, which will store the taint and capabilities of processes. A potential solution would be to implement a request—response model via a custom kernel interface for any queries, though the performance impact would be severe, requiring additional context switches. Instead, the approach adopted creates an abstract interface that purposefully removes the minutiae of the underlying system. Any solution must be trustworthy and safe, and malicious entities must not be able to exploit any *eventually consistent* components. [51]

As a final comment, it must be noted that SGX is not without its flaws; § 5.X discusses this and its impact on the project.

4.3 The Citadel IFC Model

Before work on the final Citadel implementation began, we constructed a formalisation describing the distributed nature of its design. A model helps reason about the safety and correctness of the final system, and provides the notation to properly discuss its features. Our model, which will now be presented, directly extends the one presented in § 2.1.3.

4.3.1 Reservations

Previously we had defined the concept of a *safe flow*, $A \rightarrow B$, which underpins the heart of our IFC restrictions. In previous works permission to perform an operation is granted while *implicitly* considering *how* the flow is to take place (4.1). An isolated *enforcement* component does not understand the concept of *flows*, forcing policy decisions to be defined *explicitly*; Citadel uses *reservations* for this purpose (4.2). This distinction is simple but very important when introducing *laziness* and other optimisations between the two halves of the reference monitor.

$$\mathit{operation} \rightarrow \boxed{\mathit{reference\ monitor}} \xrightarrow{\mathit{decision}} \{0, 1\} \tag{4.1}$$

$$\mathit{operation} \rightarrow \boxed{\mathit{policy} \xrightarrow{\mathit{reservation}} \mathit{enforcement}} \xrightarrow{\mathit{decision}} \{0, 1\} \tag{4.2}$$

Let Ω be the set of all operations mediated by the reference monitor, including, for example, `file_read` or `socket_open`. Also, let us define $\mathcal{R}$, the set of all *reservations*, as follows.[4]

$$\mathcal{R} = \mathcal{T} \times \wp(\Omega)$$

[4] Recalling that $\mathcal{T}$ is the set of all tags.

Further, we define a shorthand, t^α;

$$r \in \mathcal{R} \; . \; (r = (t, \alpha) = t^\alpha \implies t \in \mathcal{T} \; \wedge \; \alpha \subseteq \Omega)$$

We introduce, for a process A, additional state; $A_r \subseteq \mathcal{R}$, the set of all reservations it holds. Once a decision has been made, it is important for a reference monitor to be able to change it, revoking access if required. Thus we specify a notion of *validity* with an indicator function, $\mathcal{V} : \mathcal{R} \mapsto \{0, 1\}$. A reservation can only be used to obtain access to a resource if it is valid; invalid reservations are discarded.

4.3.2 Permissible Operations

Satisfiability To determine whether an operation may be permitted, the *constraint reservation* representing it is compared against reservations held by the process. As an example, $t^{\{\texttt{file\_read}\}}$ is the constraint for reading a file tagged with t.

A constraint τ^x is said to be satisfied by a reservation τ^y ($\tau^x \precsim \tau^y$) if the tags match, the reservation is valid, and y permits *at least* the required form of access (4.3).

$$\sigma^\alpha, \tau^\beta \in \mathcal{R} \; . \; (\; \sigma^\alpha \precsim \tau^\beta \iff \sigma = \tau \; \wedge \; \alpha \subseteq \beta \; \wedge \; \mathcal{V}(\tau^\beta) \;) \tag{4.3}$$

From here we define a *permissible operation*, $A \xrightharpoonup{\omega} t$; process A may perform operations ω on an entity tagged with t. An operation is only permissible if the process holds a reservation explicitly granting permission (4.4).

$$A \xrightharpoonup{\omega} t \iff (\exists \; t^\alpha \in A_r \implies t^\omega \precsim t^\alpha) \tag{4.4}$$

To bridge the gap between permissible flows and operations, a final definition is required; a *specific permissible flow*, $A \xmapsto{\omega, \tau} B$, meaning that A may send information to B using operations ω, via entities tagged with τ. Thus:

$$(\exists\, \omega, \tau\ .\ A \xmapsto{\omega,\tau} B) \implies A \rightarrow B \tag{4.5}$$

$$A \xmapsto{\omega,\tau} B \implies (\exists\, \omega'\ .\ A \xrightharpoonup{\omega'} \tau\ \wedge\ \omega \subseteq \omega') \tag{4.6}$$

Together, these define the relationship between an abstract policy space ($A \rightarrow B$, § 4.4.2) and concrete implementation ($A \xrightharpoonup{\omega} \tau$, § 4.4.1). As (4.6) suggests, a policy decision may grant a greater set of permissions than asked for — e.g. allowing both read and write when only write was explicitly requested. [6, 52]

A handful of small updates are required to make the existing rules consistent with the new: reservations are not transferred when creating a new entity (4.7), and reservations are not affected by capabilities as they represent a centralised component of the DIFC system.

$$A \Rightarrow B \implies A_s = B_s\ \wedge\ A_i = B_i\ \wedge\ B_r = \varnothing \tag{4.7}$$

4.3.3 Transient Entities

Alongside active and passive entities, we introduce a third type; *transient* entities. These are passive entities that are privately held by an owning active entitiy; they will be used to model Linux functions such as `pipe()` and unclaimed tainted files.

To facilitate this, all processes will be assigned a unique tag $p \in \mathcal{T}$, and any files it creates will initially also be tagged with p. Using $\mathcal{P}$ as the set of all process identifiers, we define $\mathcal{I}$ as the function returning a process's transient identifier;

$$\mathcal{I} : \mathcal{P} \mapsto \mathcal{T} \tag{4.8}$$

The modified expression for a *permissible operation* now becomes;

$$A \xrightharpoonup{\omega} t \iff \mathcal{I}(A) = t\ \vee\ (\exists\, t^{\alpha} \in A_r \implies t^{\omega} \precsim t^{\alpha}) \tag{4.9}$$

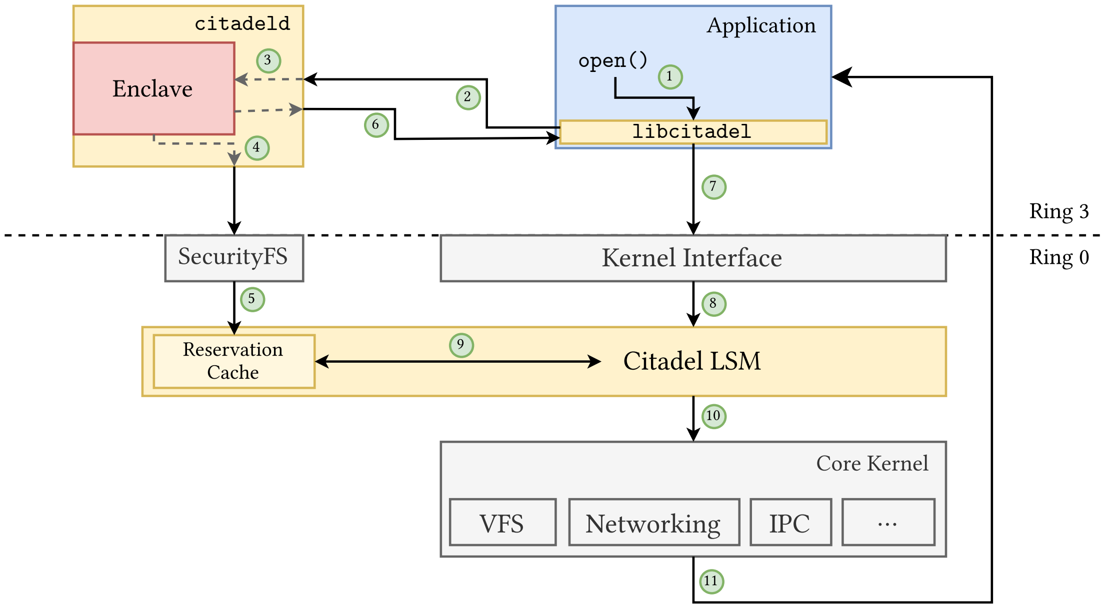

Figure 4.3: High level overview of the Citadel architecture.

4.4 Implementation

Citadel consists of three components; an LSM, `citadeld`, and `libcitadel`. Each plays an essential, symbiotic role in the operation of the reference monitor. The prototype required in excess of 9,000 lines of C and C++, and extends the Linux kernel build system (§ 4.4.6). This section shall present Citadel's architecture, guided by Figure 4.3.

Analogy The system is well modelled by the *will-call* system used by theatres and the like — clients (*processes*) reserve tickets (*permission*) to attend a show (*perform an operation*) ahead of time via phone or the internet (`citadeld`), but only receive their tickets (*reservations*) at the venue (*LSM*) on the day (*at the point of execution*).

Citadel's LSM comprise its *enforcement domain* (§ 4.4.1), and `citadeld` its *policy domain* (§ 4.4.2). Enforcement is *policy-agnostic*, implementing an abstract, tagged taint tracking system that exposes decision points to policy influence via reservations. In contrast, policy components need not be aware of the exact enforcement strategy to successfully express their protection schemes. Communication between the two domains is discussed separately in § 4.4.3.

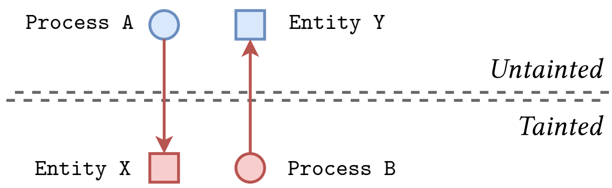

Figure 4.4: Accesses across the taint boundary taint the untainted party.

4.4.1 Enforcement

The CITADEL LSM tracks all entities within the Linux system by allocating and attaching a small data structure (< 48 bytes) to each; it computes and tracks a conservative notion of *taint* for each to ensure *safety*. Tainting in CITADEL is dynamic, meaning that entities are only policed if there is a reason. This process is *additive*, only tainting an object if it is involved in a successful operation crossing the taint boundary (Figure 4.4); this includes the child process created when a tainted process calls `fork()`. In additional to automatic propagation, taint for the majority of entities is amendable on request from the policy domain.

Recalling that entities can either be *active* or *passive*, a variety of metadata is tracked for each.

- **Active**. The only active entities in Linux are processes — these require a plethora of markers and flags, including; *taint* and its *reservation list*.
- **Passive**. There are many forms of passive entity, the most prevalent being files and other inode-backed structures. These carry *taint*, an *identifier* (tag), and an *anonymous flag*. Inode tracking is detailed first, with other types of passive entities, such as shared memory, discussed in § 4.X.[5]

4.4.1.1 Identifiers

Entities may be tagged with a single identifier; arandomly assigned 128-bit number which corresponds to a tag in the IFC model. If a security policy wishes to maintain pseudonyms for secrecy and integrity, for example, it may internally, but must convert

[5] Currently missing...

back to the system tag for enforcement.

4.4.1.2 Extended Attributes

An inode-backed entity's taint flag and identifier are copied to *xattrs* attached to it via the VFS. These occupy the `security.citadel` namespace, and are essential for ensuring that taints and identifiers persist between boots. Certain entities may be *anonymous*, as indicated by their anonymous flag, meaning that their identifier is not present as an *xattr*. This may either be because the entity does not support *xattrs* (such as files created using `pipe()`) or that the identifier is temporary (§ 4.4.1.5).

4.4.1.3 Permissions

Tainted processes must hold a valid reservation in their reservation list to perform any operation that may allow data to flow to another entity; the code for this check is attached in Appendix X for reference, but strictly follows the formal rules presented in § 4.3.2. Untainted processes bypass all checks, and thus lie outside the IFC model; the security implications of this are discussed in § 4.X.

4.4.1.4 Reservation Cache

When the system's policy enclave presents a new reservation to the LSM, it is stored in a structure called the *reservation cache*. Implemented as a red-black tree, it maps a process's identifier to a linked list of its pending reservations. This intermediary storage is necessary as LSMs are event-driven, and thus can only access an entity's state when it is presented for review. Before a permission check is carried out, the LSM ensures that the process's reservation list is up to date by;

1. *Installing pending tickets.* All reservations held for the process are moved to its internal reservation list, ready for inspection.
2. *Disposing of expired entries.* The validity function the LSM uses is time-based. When a reservation is inserted into the reservation cache, it is timestamped with an explicit expiry date — this lifetime is 15 seconds by default.

4.4.1.5 Entity Creation

As detailed in § 4.3.3, every newly spawned process is privately tagged by the LSM as if it were a passive entity. The purpose of this identifier is not to directly identify the process, but to provide a mechanism for associating any private, passive entities it creates with it. This includes the file descriptors provided by `pipe()`, and any new files it creates using `open()`. Every process always has permission to access its transient entities, and external entities can only gain the right to access them if they;

1. Are a child processes and request access to their parent, or
2. The process officially *claims* them via the policy enclave, which gives them an independent tag and removes the entity's status as transient.

`fork()` In Linux, child processes are initially exact clones of their parent, with access to the same state and file descriptors. Thus children of tainted processes are also tainted, but importantly do not assume the same rights as their parents — open file descriptors will not function without revalidation (§ 4.4.4), and children must request the right to to their parent's transient entities to use pipes or similar.[6] It is the prerogative of the policy enclave to validate that the security contexts of the parent and child have not diverged. [7]

4.4.2 Policy Components

The policy counterpart to the LSM's enforcement is contained within `citadeld`, a userspace service that hosts the core SGX enclave. `citadeld` is modular, hosting an independent policy module sitting on top of an enforcement translation library (Figure 4.5).

4.4.2.1 Abstract Policy Module

The policy module embedded in `citadeld` is presented with a simple, event-driven interface; this streamlines their implementation, allowing more emphasis to be put of

[6] Unrelated processes are unable to do this.
[7] mention leakage through fd state

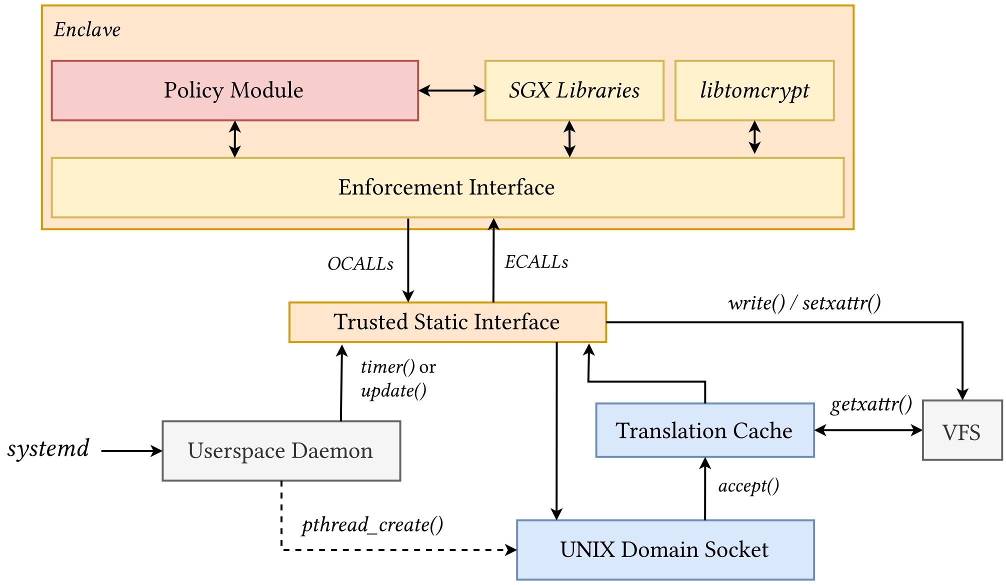

Figure 4.5: Overview of the components inside `citadeld`.

correctness. Their implementation is based around a single method, through which their permission is sought when required; `asm_handle_request(3)`.

The simplest possible policy is as follows; any operation is deemed permissible. The request parameter, amongst other things, holds the target identifier and set of operations.

```
citadel_response_t asm_handle_request (pid_t pid,
        struct citadel_op_request *request, void *metadata) {
    return CITADEL_OP_APPROVED;
}
```

This can be considered to determine the validity of an operation, $A \xrightarrow{\omega} t$, based on its knowledge of any implicated flows ($A \to *$).

Operations Entity operations, Ω, are presented as `citadel_operation_t`, a simple bit mask over the operations Citadel recognises (Appendix X). Similarly, policy decisions are represented using `citadel_response_t`; these may be *approved*, *rejected*, *error*, *invalid*, *granted*,[8] and *forged*.[9]

4.4.2.2 Host Application

Before requests are presented to the resident policy module (§ 4.4.3), various steps need to be taken in preparation. Requests often refer to absolute filepaths, creating a need to retrieve their tags, if they exist — these requests are served by the security *xattrs* attached to the VFS file. Translation is performed pre-emptively depending on the operation requested, and to minimise any lookup overhead, results are cached in a *translation cache*. This is implemented using *sparsehash*,[10] and great care is taken to detect stale entities that may confuse the internal decision process.

4.4.2.3 Enforcement Interface

The policy module is interchangeable, but the enforcement interface acts as the backbone of the enclave. All requests are routed through it as a sanitsation step, detecting forgery or and invalid data, and all information leaving it is formatted and signed[11] as appropriate. The process of installing reservations created by the enforcement interface, on behalf of the policy module, is detailed in § 4.4.3.

4.4.2.4 `libtomcrypt`

We ported *libtomcrypt*,[12] a leading open source cryptography library, to function inside an SGX enclave. This was necessary to support a number of the encryption mechanisms the system requires, on top of those already provided by SGX. This was achieved by replacing its backing precision arithmetic library to be an SGX-aware version of *GMP*,[13] and forcing it to statically allocate its memory (as SGX v1 lacks support for dynamic memory management). Further changes rewrote the internal random number generator to use the one provided by SGX, and rework its exception strategy to remove `abort()`, an illegal instruction inside an enclave.

[8] Approved, and confirming that the process is recognised as the owner of the entity.
[9] Discussed in § 4.4.5.
[10] `https://github.com/sparsehash/sparsehash`
[11] Encryption is discussed in § 4.4.3
[12] `https://github.com/libtom/libtomcrypt`
[13] `https://github.com/intel/sgx-gmp`

4.4.3 Communication Pathways

There are three notable I/O pathways between components within Citadel;

1. Applications ⟷ Policy Enclave.
   All application requests (via `libcitadel`, § 4.4.4) are sent to the policy enclave using a standard domain socket; `/run/citadel.socket`. To ensure that all processes have the right to communicate with the reference monitor, a special tag, $\tau = 2^{128} - 1$, is assigned to it, asserting the reference monitor's ownership of it and whitelisting it in the LSM.

2. Policy Enclave ⟷ LSM.
   These parties communicate using two mediums; *SecurityFS* and *xattrs*. All messages between these are encrypted using AES-256-GCM [53, 54]; the key is chosen during the system's initialisation (§ 4.4.3.1).
   Reservations are installed using a custom *SecurityFS* interface,[14] and are synchronously inserted into the reservation cache. The policy enclave may decide to invoke an operation directly on a file; this is handled by invoking `setxattr()`, which the LSM intercepts, triggering it to enact the required changes to the entity; the most common use of this method is entity tagging.

3. LSM ⟶ Applications.
   To verify their identities with the policy enclave, applications need to present a *ptoken* with each request; this process is described in § 4.4.5.1, but can be generated by reading from a globally readable *SecurityFS* interface.[15]
   In addition to this, `libcitadel` occasionally needs to check the tag associated with a path or file descriptor; this is managed using the existing `libc` *xattr* methods.

4.4.3.1 Initialisation and Encryption

Whenever the system boots, the LSM is first to come online — `citadeld` may start at any time beyond this, meaning that the LSM must be capable of operating independently. In this case, when the LSM is isolated, the system will tend towards a state of

[14] `/sys/kernel/security/citadel/update`
[15] `/sys/kernel/security/citadel/ptoken`

complete lockdown (for tainted processes). Thus the mechanism by which the LSM and policy enclave initialise communication is vital for secure operation; Citadel achieves this with a pair of 2048-bit RSA keypairs, one for the enclave ($E_{P/S}$) and one for the kernel ($K_{P/S}$).

SGX does not provide protection against reverse engineering, thus the enclave's keys must be provided to it as a sealed entity; sealing here uses `MRSIGNER`, allowing any policy enclave provided by the *sealing authority* to fully function, and is compiled into the kernel, available via *SecurityFS*.[16]

Once a policy enclave has been initialised it must verify itself; the LSM issues a random challenge[17] encrypted using the enclave's public key, and expects a reply using the corresponding private key.

$$\text{LSM} \rightarrow \text{Enclave} : \{\textit{challenge}\}_{E_P}$$
$$\text{Enclave} \rightarrow \text{LSM} : \{\textit{challenge}, \textit{PID}, \textit{identifier}, \textit{aes\_key}, ...\}_{E_S}$$

Given E_S is only held sealed, any entity providing a valid challenge is trusted and considered part of the Citadel TCB. The challenger's PID is stored to detect any adversarial replay messages. RSA is only used for this initial exchange; it would be too slow to use for all messages. Thus the AES key provided in the response forms the basis of all future communication.

Citadel uses AES to protect sentitive messages as every SGX-capable processor supports *AESNI* [55] to accelerate AES in hardware. SGX provides this natively inside enclaves, and the official *Intel AESNI* driver is included in the mainline kernel. *AESNI* provides an encryption bandwidth in excess of 1 Gbps,[18] far exceeding the capacity required in this system, and thus adding negligable overhead. The system's AES key updates with every message sent from the enclave using an SGX-approved source of

[16] `/sys/kernel/security/citadel/sealed_keys`

[17] `/sys/kernel/security/citadel/challenge`

[18] Our experiments on the evaluation hardware exceeded this, but this value is fair given the large range of processors that support SGX.

entropy;[19] this adds minimal overhead and constitutes good practice. A copy of the initial key presented in the challenge response is retained and used in cases when a static key is essential.

4.4.4 `libcitadel`

CITADEL provides a userspace auxiliary library to make integrating existing programs as effortless as possible. For each mediated *syscall* (e.g. `open()`), it provides a proxy function (`c_open()`): this would, in a future iteration of this design, be integrated directly into `libc`, but currently requires no major changes to applications' workflows. A good example of this in action is the ported version of *Nginx* (§ 5.X).

`libcitadel` performs two main functions;

1. Communication with the policy enclave.
2. Tracking and predicting what permissions it believes the process has.

Communication is facilitated via the UNIX domain socket provided by `citadeld`. A zero-copy approach[20] is used to minimise latency on both sides; this is optimised for in the protocol design, and great effort has been put into minimising the cost of communication. Each communicant verifies the PID of the other party, as detailed in § 4.4.5.1.

Caching at this level has a tremendous impact on overall performance. When reading a large file, for example, a program may make thousands of calls to `c_read()` on the same file — making external calls to `citadeld` would be wasteful as processes have, in most cases, enough information to infer their current position.

To this end, every process maintains a list of *expectations* — the reservations it believes it has, including their validity — and whether it has inferred that it is tainted. They cannot exactly know the true values of these, especially as the policy enclave may grant different permissions than asked for, but in *Nginx*, as an example, over 97%

[19] $new \leftarrow old \oplus update$

[20] Of course excluding copying in the kernel and when transferring the request into the enclave.

```c
int c_open(const char *pathname, int oflag, mode_t mode) {
    int fd; bool from_cache = false;
    bool creating = access(pathname, F_OK) < 0 && (oflag & O_CREAT) > 0;

    // Pre-emptively attempt access if I suspect I'm not tainted.
    // Alteratively, register a transient file if we're creating it.
    // -- close and reopen to ensure it is independently tagged.
    if (!am_tainted() || creating) {
        fd = open(pathname, oflag, mode);
        if (!am_tainted() && fd > 0)
            return fd;
        if (fd == -1 && errno != -EPERM)
            return -1;
        if (fd != -1)
            close(fd);
    }

    // Request access from the policy enclave. Claim file if not tagged.
    if (!citadel_file_open(pathname, strlen(pathname)+1, &from_cache))
        return -EPERM;

    // Continue as normal.
    fd = open(pathname, oflag, mode);
    citadel_declare_fd(fd, CITADEL_OP_OPEN);
    if (!am_tainted()) set_taint();
    return fd;
}
```

Listing 1: The `libcitadel` shim function for `open()`.

of requests were servable locally in a realistic workload. Using the same workflow, untainted processes speculatively execute operations, again removing the need to involve `citadeld`. The performance gain of requests served from the cache reduces the overhead from $\mathcal{O}(10\mu s)$ to $\mathcal{O}(100ns)$.

A core challenge of the cache is relating open file descriptors to the permissions they require. This requires some manual work, including fetching its *xattr* tag with `fgetxattr()` and estimating the expiration time of the LSM's underlying reservation. Ideally `libcitadel` requests revalidation just before expiry to ensure no unexpected drop in service; this is particulary important for applications unaware of Citadel.

Special care has to be taken when handling child processes. As already discussed, children are given a copy of all parent process state — this includes its `libcitadel` cache. Although the LSM does not pass any reservations on to the child, `libcitadel` maintains the same *expectation* cache. Entries in it are marked as invalid, to force the child to revalidate a file descriptor before first use. Additionally, we assume that a process trusts the initialisation code of its child, enabling `libcitadel` to delete the parent's *ptoken* (§ 4.4.5.1); the `c_fork()` function handles this automatically.

4.4.5 Additional Security Features

Citadel implements a handful of additional security mechanisms to reinforce potentially vulnerable aspects of the system. Both the policy enclave and LSM use a process's PID as its primary identifier — Citadel implements two schemes to protect this notion of identity and help prove it.

4.4.5.1 *ptokens*

Before a process may interact with `citadeld`, it must retrieve its *ptoken* from the LSM.[21] The purpose of this record (4.10) is twofold;

a. Inform `libcitadel` about the process's metadata in the eyes of the LSM, and

b. Provide an authenticable access token to present to `citadeld`, verifying the process's identity. This is AES encrypted using K, the system's designated static AES key, which is unknown to the process.

$$\textit{ptoken} \rightarrow (\text{citadel\_pid}, \text{identifier}, \text{token}, \{\text{identifier}, \text{token}, \text{pid}\}_K) \tag{4.10}$$

Whenever a process connects to the `citadeld` socket, its identity is retrieved from the underlying transport mechanism. At both the sender and receiver the identity of the other is verified using this method, and additionally `libcitadel` expects the decrypted *token*, a randomly generated byte-array, to be returned by `citadeld`, inspiring confidence that the response has not been forged.

[21] Read from `/sys/kernel/security/citadel/ptoken`

```c
// Get PID of sender.
struct ucred cred;
socklen_t len = sizeof(struct ucred);
getsockopt(socket, SOL_SOCKET, SO_PEERCRED, (void*)&cred, &len);
uint64_t pid = cred.pid;
```

4.4.5.2 PID Protection

The LSM also implements a mechanism to detect PID forgery — as shown in Appendix A, it is theoretically possible for a process to modify its PID with the help of a malicious kernel module. This, if unchecked, would be detrimental for the LSM's integrity, as it would allow a process to silently assume the identity of another. To this end, the LSM stores a process's PID within its security structure and routinely checks to ensure it does not change unexpectedly.[22] Any process deemed to have an illegitimate PID is denied access to all entities, effectively killing it.

4.4.6 Citadel Build System

Building Citadel requires both the kernel and policy enclave to be in agreement about the RSA keys to be used; without this building will fail. This is achieved using a preparatory script that does the following;[23]

1. Generate two *OpenSSL*[24] 2048-bit RSA keys in *DER* format — the kernel's *Crypto API* requires keys to present themselves as *ASN.1 structures*.[25]

2. Compile and launch Citadel's *preparatory enclave*; this must be signed with the same *signing identity* as any policy enclaves generated. This ingests the two keys and generates a sealed keyset to be presented to initialising enclaves.

3. An interface file (`keys.h`) is then generated in the kernel's source directory — this file contains the kernel's keypair, the enclave's public key, and the aforementioned sealed keyset. The key files are now deleted.

[22] A valid change would be on `fork()`, in which case the stored PID should equal the parent process's.
[23] The script assumes a valid kernel with the Citadel LSM installed is present.
[24] `https://www.openssl.org/`
[25] `https://tls.mbed.org/kb/cryptography/asn1-key-structures-in-der-and-pem`

4. The policy enclaves are then built and signed. The kernel may now be compiled.

Chapter 5

Evaluation

Three core questions hang over CITADEL's viability — its security, expressivity, and performance. This chapter presents a thorough investigation of the prototype's performance profile, a discussion around its security implications, and an illustrative example in favour of its expressability.

5.1 Performance

The CITADEL prototype demonstrates highly impressive performance, matching, and in places surpassing, related approaches, despite the architectural disadvantage it is at. We present its behaviour relative to native Linux kernel in three ways;

1. Application-level microbenchmarks, tracing the duration of *syscalls* both natively and through `libcitadel`. (§ 5.1.2)
2. IPC bandwidth microbenchmarks in both *intra-* and *inter*-process contexts. (§ 5.1.3)
3. Real-world NGINX performance benchmarks for both low-latency and high-bandwidth configurations. (§ 5.1.4)

The results presented here are best compared to *Flume* [6] — *CamFlow*, although implemented similarly, has a far greater scope that this project.

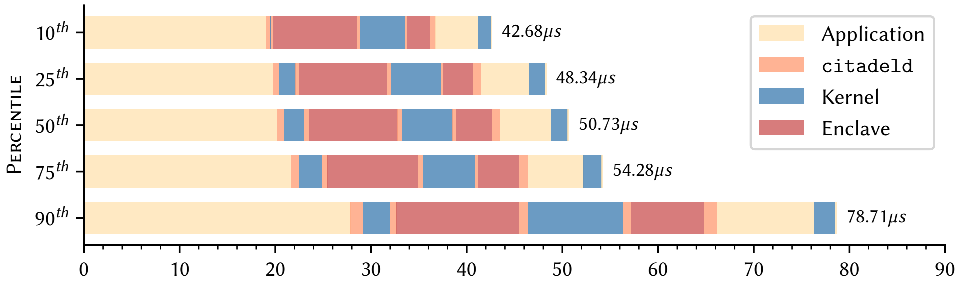

Figure 5.1: Control flow inhabitation for `libcitadel`'s `c_open()` function, $n = 100$.

5.1.1 Evaluation Environment

The research machine used for evaluation contained a quad-core Intel® Core™ i5-6600 (which supports SGX v1), 16 GiB RAM, and a 1-Gbps NIC. The primary disk provided 389 MBps read and 210 MBps write.[1] For all experiments running under Citadel, `citadeld` was running via `systemd` — all debugging tools were disabled and the enclave was built in hardware pre-release mode. Linux v5.6.0 was the base kernel for all trials; Citadel adds its LSM on top of this.

Both Tables 5.1 and 5.2 report the sample mean and standard deviation. Figures 5.2, 5.3, and 5.4 plot the sample medians and interquartile range for each point. The Wilcoxon paired signed rank test was chosen to determine statistical significance. [56]

5.1.2 *syscall* Microbenchmarks

A custom benchmark tool was built to assess the overall impact Citadel has on *syscall* performance — for example, the duration of `open()` compared to `c_open()`. Table 5.1 presents these results. To give a fair comparison, two figures are reported for Citadel. *Amortised* refers to the normal operation of `libcitadel`, in which the majority of queries are served from the cache; overhead arises from both local cache operations and added kernel latency from the LSM. The other figure, *Cache Miss*, gives the overhead when caching is disabled, thus including communication with `citadeld`.

[1]As reported by the dd tool.

	Native	Citadel	
		Amortised	*Cache Miss*
`open()`	1.675 ± 0.076	6.083 ± 0.129	50.133 ± 1.482
`read()`	5.724 ± 0.206	7.010 ± 0.192	54.736 ± 1.556
`write()`	14.340 ± 0.208	15.597 ± 0.250	63.824 ± 1.902
`close()`	0.651 ± 0.005	0.718 ± 0.011	
`socket()`	1.446 ± 0.179	3.156 ± 0.291	
`bind()`	0.762 ± 0.023	1.911 ± 0.183	49.110 ± 1.746
`listen()`	0.705 ± 0.015	1.882 ± 0.149	48.411 ± 1.386
`connect()`	16.570 ± 0.278	17.961 ± 0.330	66.273 ± 2.147
`shmget()`	1.880 ± 0.122	1.913 ± 0.111	49.326 ± 1.466
`shmat()`	0.420 ± 0.005	1.575 ± 0.134	47.997 ± 1.560
`shmctl()`	0.418 ± 0.005	0.743 ± 0.083	45.912 ± 1.114
`shmdt()`	0.415 ± 0.003	1.342 ± 0.040	
`pipe()`	1.110 ± 0.061	1.288 ± 0.069	47.334 ± 1.147
`mkfifo()`	3.865 ± 0.048	11.509 ± 0.405	59.623 ± 1.788
`fork()`	47.866 ± 3.175	48.647 ± 3.457	81.174 ± 3.829
`citadel_init()`	–	0.801 ± 0.009	34.940 ± 1.329

Table 5.1: `libcitadel` microbenchmarks.
All values are in μs and the sample standard deviation is shown alongside the mean. For Citadel, both the amortised and average cache-miss durations are given. Only one value is given if the operation is not affected by a cache miss. $n = 10^6$.

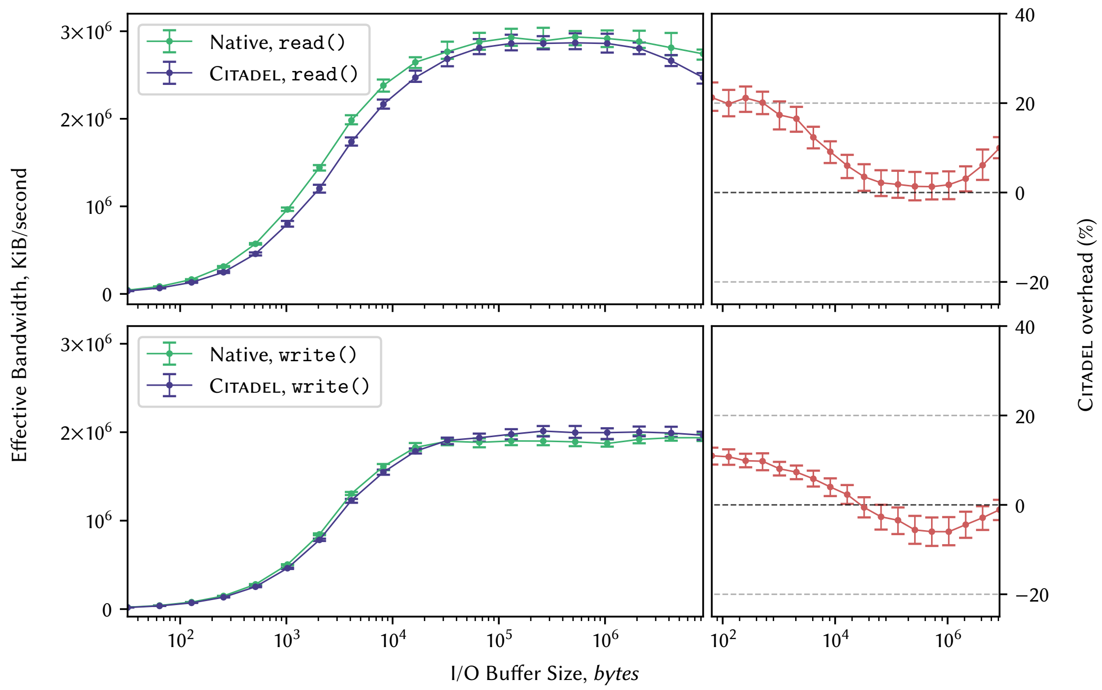

Figure 5.2: Effective `read()/write()` bandwidths for both the native Linux kernel and CITADEL. The percentage overhead is also presented. $n = 200$ per buffer size.

Overall `libcitadel` contributes $\sim 1\mu s$ of overhead (amortised) on average — this rises to $\sim 40\mu s$ on a cache miss. Figure 5.1 presents a more detailed view of where exactly this overhead arises, approximately plotting where the control flow for `c_open()` moves (on a cache miss). Interestingly, the slowest component is the communication channel between `libcitadel` and `citadeld` ($26\mu s$ median);[2] as a result, the core reference monitor functionality only adds a median penalty of $24\mu s$. The final *Kernel* call before terminating is the internal call to `open()`. Additionally, the 10^{th} percentile demonstrates that the first *Kernel* call is not always required if the entity's metadata is resident in the cache.

Figure 5.2 plots observed effective bandwidth whilst reading from and writing to a 16 MiB file with different sized buffers. The benchmark driving this was adapted

[2]Included in the *Application* regions.

for Linux from one written by R. Watson for FreeBSD. [57] The results clearly show CITADEL having a more adverse effect on performance for smaller buffer sizes; this is logical, as smaller buffers force a larger number of calls to `read()`/`write()`. The reason for CITADEL providing better performance for large buffers with `write()` is unclear — the difference is statistically significant (at 5% confidence) and reproducable. More work is required to narrow down the root causes, but hypotheses include the fact that regular, small delays could ease pressure on microarchitectural cache in the processor, affording slight optimisations.

5.1.3 IPC Microbenchmarks

Again using the modified Watson benchmark, we investigated the effect CITADEL has on end-to-end IPC performance. We investigate *pipes*, *local sockets* (`socketpair()`), and *regular sockets*, between 2 threads (Figure 5.3) and between a parent and child process (Figure 5.4).

Overall the results between the two contexts are similar — both see approximately 20% degradation in the worst cases, tending towards equal performance when using $\sim 10^5$-byte blocks. At first glance it would appear that CITADEL affects the performance between 2 threads slightly more than 2 processes, but in fact the truth is that CITADEL performs nearly identically in both. The native performance is more heavily optimised when sending between two threads; any latency gained by the kernel is overshadowed by CITADEL.

In a similar way to `write()`, CITADEL outperforms the native kernel in both contexts using `pipe()`. The readings are noisy, but statistically significant (at 5%) and reproducable. The causes is again unknown, but noting that the native kernel's throughput halves after buffers of 8 KiB, we highly suspect that this is the result of cache exhaustion or inopportune paging. It is also notable that readings for CITADEL exhibit a far larger interquartile range for large buffer sizes than the native kernel, a side effect that we see repeated in real-world testing.

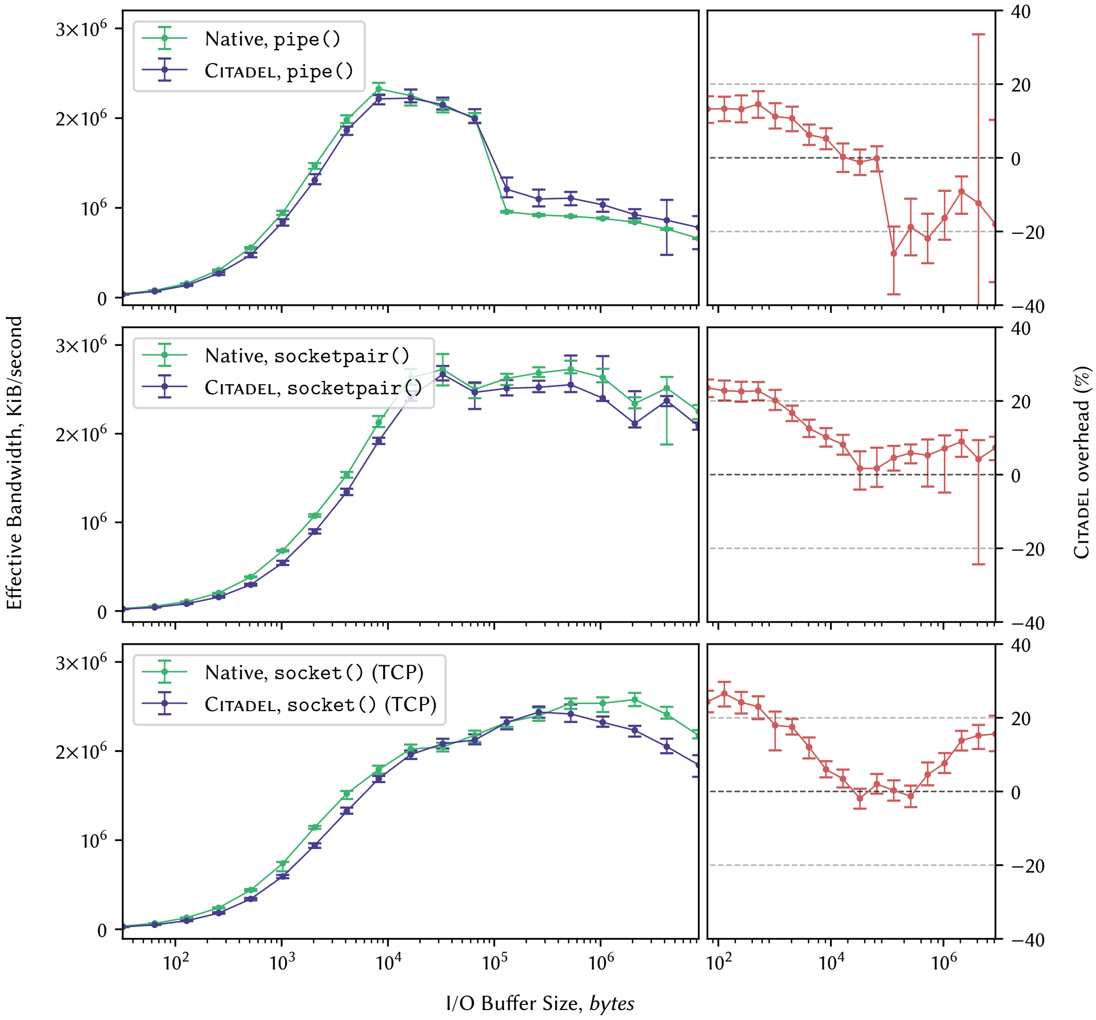

Figure 5.3: Effective bandwidths for various types of IPC between *2 threads*, $n = 200$.

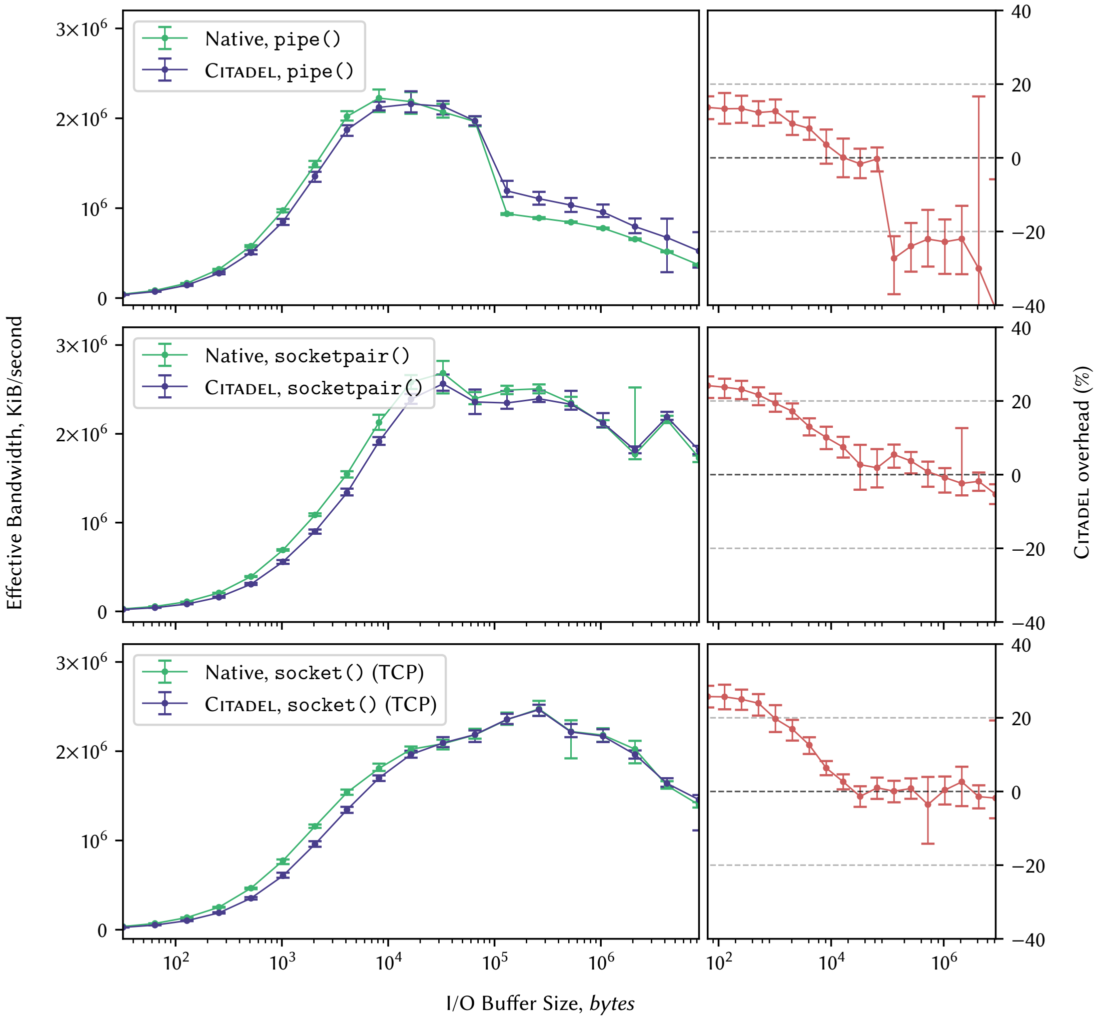

Figure 5.4: Effective bandwidths for various types of IPC between *2 processes*, $n = 200$.

	Native	Citadel		
		Untainted	*Tainted*	*Overhead*
Webserver Benchmark, *100-byte packets*				
Latency	$35.73\mu s$	$36.18\mu s$	$44.35\mu s$	24%
– *std. dev.*	$13.85\mu s$	$14.12\mu s$	$13.26\mu s$	
– *max.*	$536\mu s$	$554\mu s$	$508\mu s$	
Requests/s	$2.748 \cdot 10^4$	$2.717 \cdot 10^4$	$2.214 \cdot 10^4$	19%
Bandwidth	177.28 Mbps	168.72 Mbps	143.04 Mbps	18%
10GB File Transfer				
Bandwidth	1.404 Gbps	1.410 Gbps	1.413 Gbps	$\sim 0\%$
– *std. dev.*	0.428 Gbps	0.440 Gbps	0.549 Gbps	
Duration	$56.98s$	$56.74s$	$56.62s$	$\sim 0\%$
– *std. dev.*	$19.45s$	$18.97s$	$23.63s$	

Table 5.2: Nginx performance comparinson between native Linux, and both untainted and tainted Citadel, $n = 25$.

5.1.4 Nginx Benchmarks

To validate the performance results presented thusfar, we ported the entirety of the Nginx webserver[3] to function inside Citadel. No optimisations were made to the codebase — the only changes were changing core `libc` function calls to use their `c_*` `libcitadel` counterpart.

Two trials were run; a low-latency benchmark[4] and a 10GB HTTP file transfer (high-bandwidth). The webserver was configured to only run a single server process to ensure it was exercised to its full extent, and was setup to use the *loopback* interface[5] to eliminate any interference from outside the OS.

[3]`https://www.nginx.com/`
[4]`https://github.com/wg/wrk`
[5]`http://127.0.0.1/`.

The results, displayed in Table 5.2, are not suprising. For the low latency tests we observe the same 20-25% overhead as seen from TCP sockets in § 5.1.3 using the same buffer size. The high-bandwidth tests show CITADEL performing equally as the native kernel, only differing in that it had a larger sample standard deviation. The large file transfer is also interesting as it is the first time we see file descriptor revalidation happening automatically in the wild on `read()` and `write()` being called.

Chapter 6

Summary and Conclusions

Appendix A

PID Tampering: Proof of Concept

```c
static void* retrieve_symbol(const char *sym) {
   static unsigned long faddr = 0;

   // Compare kernel symbol with query.
   int symcmp(void* data, const char* sym, struct module* mod,
              unsigned long addr) {
      if(!strcmp((char*)data, sym)) {
         faddr = addr;
         return 1;
      }
      else return 0;
   };

   kallsyms_on_each_symbol(symcmp, (void*)sym);
   return (void*)faddr;
}


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 = retrieve_symbol("change_pid");
    _alloc_pid = retrieve_symbol("alloc_pid");
    /* ... */
    /* On SysFS call execute change_pid(void) */
    return 0;
}
```

Listing 2: Outline for a proof of concept kernel module to change a process's PID.

Bibliography

[1] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. volume 3302, pages 129–145, 11 2004. (Cited on page 3.)

[2] David Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. 1973. (Cited on page 3.)

[3] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, 1976. (Cited on pages 3 and 4.)

[4] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Workshop on Computer Security Foundations*, CSFW '99, page 228, USA, 1999. IEEE Computer Society. (Cited on page 4.)

[5] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Symposium on Cloud Computing (SoCC'17)*. ACM, 2017. (Cited on pages 4, 5, and 21.)

[6] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, October 2007. (Cited on pages 4, 5, 21, 31, and 45.)

[7] Department of Defence. *Trusted Computer System Evaluation Criteria*, August 1983. (Cited on page 4.)

[8] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*, pages 186–197, 1998. (Cited on page 4.)

[9] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 2000. (Cited on page 4.)

[10] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 129–142, New York, NY, USA, 1997. Association for Computing Machinery. (Cited on page 4.)

[11] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery. (Cited on page 8.)

[12] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery. (Cited on pages 8 and 9.)

[13] Ittai Anati, Shay Gueron, Simon Paul Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. 2013. (Cited on pages 8 and 14.)

[14] Intel®. *Intel® 64 and IA-32 Architectures. Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*, September 2015. (Cited on pages 8 and 10.)

[15] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, New York, NY, USA, 2015. Association for Computing Machinery. (Cited on page 9.)

[16] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3), August 2015. (Cited on page 9.)

[17] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1570–1581, New York, NY, USA, 2015. Association for Computing Machinery. (Cited on page 9.)

[18] Intel®. *Intel® Software Guard Extensions SDK*, March 2020. (Cited on page 9.)

[19] Shay Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016. `https://eprint.iacr.org/2016/204`. (Cited on page 9.)

[20] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, page 640–656, USA, 2015. IEEE Computer Society. (Cited on page 10.)

[21] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016. (Cited on page 10.)

[22] Intel®. *Intel® Software Guard Extensions Programming Reference*, October 2014. (Cited on pages 14 and 15.)

[23] Intel®. *Intel® SGX: Intel® EPID Provisioning and Attestation Services*, March 2016. (Cited on pages 14 and 16.)

[24] Ernie Brickell and Jiangtao Li. Enhanced privacy id: A direct anonymous attestation scheme with enhanced revocation capabilities. volume 9, pages 21–30, 01 2007. (Cited on page 16.)

[25] The Trusted Computing Group. *Trusted Platform Module (TPM) Summary*, January 2008. (Cited on page 19.)

[26] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 17–30, New York, NY, USA, 2005. Association for Computing Machinery. (Cited on page 22.)

[27] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 263–278, USA, 2006. USENIX Association. (Cited on page 22.)

[28] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 293–308, USA, 2008. USENIX Association. (Cited on page 22.)

[29] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 12, USA, 2012. USENIX Association. (Cited on page 22.)

[30] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. *SIGPLAN Not.*, 44(6):63–74, June 2009. (Cited on page 22.)

[31] Oracle®. *The Java® Virtual Machine Specification*, February 2015. (Cited on page 22.)

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004. (Cited on page 22.)

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. (Cited on page 22.)

[34] Lars Richter, Johannes Götzfried, and Tilo Müller. Isolating operating system components with intel sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, New York, NY, USA, 2016. Association for Computing Machinery. (Cited on page 23.)

[35] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017. (Cited on page 23.)

[36] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. Delegatee: Brokered delegation using trusted execution environments. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 1387–1403, USA, 2018. USENIX Association. (Cited on page 23.)

[37] J. B. Djoko, J. Lange, and A. J. Lee. Nexus: Practical and secure access control on untrusted storage platforms using client-side sgx. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 401–413, 2019. (Cited on page 23.)

[38] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 689–703, USA, 2016. USENIX Association. (Cited on page 24.)

[39] Docker. *Introduction to Container Security: Understanding the isolation properties of Docker*, August 2016. (Cited on page 24.)

[40] 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*, pages 38–54, 2015. (Cited on page 24.)

[41] F. Kelbert, P. Pietzuch, and J. Crowcroft. *Maru: SGX-Spark Deep Dive*, November 2017. (Cited on page 24.)

[42] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 533–549, Savannah, GA, November 2016. USENIX Association. (Cited on page 24.)

[43] Google. *NaCL: Native Client.* (Cited on page 24.)

[44] J. Anderson. *Computer Security Technology Planning Study — Volume 2*, October 1972. (Cited on page 25.)

[45] C. Irvine. The reference monitor concept as a unifying principle in computer security education. April 2001. (Cited on page 25.)

[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018. (Cited on page 26.)

[47] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 319–334, USA, 2015. USENIX Association. (Cited on page 26.)

[48] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery. (Cited on page 27.)

[49] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. *SIGOPS Oper. Syst. Rev.*, 31(5):52–65, October 1997. (Cited on page 27.)

[50] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21:9–21, 1988. (Cited on page 27.)

[51] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. (Cited on page 28.)

[52] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. pages 293–308, 01 2008. (Cited on page 31.)

[53] Joan Daemen and Vincent Rijmen. The block cipher rijndael. volume 1820, pages 277–284, 01 1998. (Cited on page 38.)

[54] David A. McGrew and John Viega. The galois/counter mode of operation (gcm). 2005. (Cited on page 38.)

[55] Shay Gueron and Intel®. *Intel® Advanced Encryption Standard (AES) New Instructions Set*, May 2010. (Cited on page 39.)

[56] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. (Cited on page 46.)

[57] Robert N. M. Watson. *Advanced Operating System — L41, University of Cambridge*, March 2020. (Cited on page 49.)