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

October 1, 2021

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.

Signed: *A.H. Bell-Thomas*

Date: *October 1, 2021*

Abstract

Information Flow Control (IFC) is a powerful tool for protecting data in a computer system, enforcing not only who may access it, but also how it may be used throughout its lifespan. Intel's Software Guard Extension (SGX) affords complementary protection, providing a general-purpose Trusted Execution Environment for applications and their data. To date, no work has been conducted considering the overlap between the two, and how they may mutually reinforce each other.

This dissertation presents Citadel, a modular, SGX-backed reference monitor to securely and verifiably implement IFC methods in the Linux kernel. Its prototype externalises policy decisions from its enforcement security module, providing a userspace promise-of-access model with asynchronous fulfillment. By aliasing system calls, the system transparently integrates with unmodified applications, and amortises the performance cost of integration by inferring processes' underlying security contexts.

Observed results are promising, demonstrating a worst-case median performance overhead of 25%. In addition, the Nginx webserver is demonstrated running under Citadel; high bandwidth transfers exhibit near parity with the native Linux kernel's performance. This work illustrates the potential viability of a symbiotic enclave-kernel relationship for security implementations, something that may, in the long run, benefit both.

Contents

Figures, Tables, and Listings ii

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 4
2.1.3 Modelling 5
2.2 Intel® SGX 8
2.2.1 Security Characteristics 9
2.2.2 Architecture and Implementation 9
2.2.3 Enclave Lifecycle 11
2.2.4 Attestation 13
2.2.5 Sealing 15
2.2.6 SGX Versions 15
2.3 Aspects of the *Linux* Kernel 16
2.3.1 Virtual File System 16
2.3.2 Linux Security Modules 17

3 Related Work **19**
3.1 *Flume* and *CamFlow* 19
3.1.1 Other IFC Systems 20
3.2 Interoperation between Linux and SGX 20
3.3 Dataflow Protection using SGX 21

4 Citadel **23**
4.1 Motivation 23
4.2 Challenges 24
4.3 The Citadel IFC Model 28
4.3.1 Reservations 28

4.3.2 Permissible Operations . . . 29
4.3.3 Transient Entities . . . 30
4.4 Implementation . . . 31
4.4.1 Enforcement . . . 32
4.4.2 Policy Components . . . 34
4.4.3 Communication Pathways . . . 37
4.4.4 `libcitadel` . . . 39
4.4.5 Additional Security Features . . . 41
4.4.6 Citadel Build System . . . 42

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
5.2 Security . . . 53
5.2.1 Citadel TCB . . . 53
5.2.2 IFC Model Implications . . . 55
5.2.3 SGX Vulnerabilities . . . 56
5.3 Use Cases . . . 56

6 Conclusions **59**

A PID Tampering: Proof of Concept **61**

Acronyms **63**

Figures, Tables, and Listings

2.1 Overview of the four core IFC events 5
2.2 Abstract overview of SGX's protection in an adversarial environment. . 8
2.3 A high-level overview of the SGX hardware and software architecture. 10
2.4 The process of creating and initialising an enclave. 12
2.5 Overview of notable SGX x86 instructions in an enclave's lifecycle. . . 14

4.1 Abstract *syscall* control flow route for enclave integration. 25
4.2 Two possible enclave integration designs. 26
4.3 High level overview of the Citadel architecture. 31
4.4 Accesses across the taint boundary 32
4.5 Overview of the components inside `citadeld`. 35
4.6 The `libcitadel` shim function for `open()`. 40
4.7 PID retrieval from an active domain socket. 42

5.1 Control flow inhabitation for `libcitadel`'s `c_open()` function, $n = 100$. 46
5.2 `libcitadel` microbenchmarks 47
5.3 Effective `read()`/`write()` bandwidths for both the native Linux kernel and Citadel. 48
5.4 Effective bandwidths for various types of IPC between *2 threads*. 50
5.5 Effective bandwidths for various types of IPC between *2 processes*. . . . 51
5.6 Nginx performance comparinson between native Linux, and both untainted and tainted Citadel. 52

A.1 Expose unexported symbols from the global namespace using `kallsyms`. 61
A.2 Exploit unexported symbols to change the PID of the current process. . 62

Chapter 1

Introduction

Defending computer systems against malicious programs and enforcing the isolation of protected components has always been exceedingly challenging. A system's *Trusted Computing Base (TCB)*, defines the minimal set of components critical to establish and maintain system security and integrity. This traditionally includes, amongst others; the OS kernel; device drivers; device firmware; and hardware. Compromise of a trusted component inside a system's *TCB* directly threatens any secure application. One approach to hardening a system's security is to minimise its *TCB*, diminishing its potential *attack surface*.

An increasingly common trend is outsourcing a system's physical layer to a foreign party, for example, a *cloud provider* — this is beneficial in terms of cost and flexibility, but many security considerations assume that the physical layer itself can be trusted. This is not guaranteed when the physical layer is a *virtual machine*, inflating the system's *TCB* with an external and transparent software layer, the underlying *hypervisor*.

Trusted Execution Environments (TEEs) have long been explored by the security community as potential protection against this. They generate isolated processing contexts in which an operation can be securely executed irrespective of the rest of the system — one example is software *enclaves*. *Enclaves* are general-purpose *TEE* provided by the CPU, protecting the logic found inside at the architectural level. Intel's Software Guard

Extensions (SGX) is the most prolific example, affording a *black-box* environment and runtime for arbitrary apps to execute under.

An alternative approach is to use *Information Flow Control (IFC)* to police system components. Enforced using a *reference monitor*, IFC models permissible data use, manipulating systems at a granular level.

This work explores 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 several common situations.

Contributions

- CITADEL, a prototype implementation of a modular *reference monitor* protected using Intel SGX, empowering IFC techniques to operate with autonomy and protection from the host operating system. Enforcement is achieved using a *Linux Security Module (LSM)* 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 performance implications, featuring the *Nginx* production webserver. Worst-case performance shows a 24% decrease in request throughput, with other trials reporting performance parity with native Linux. Additionally we report a median overhead of $43\,\mu s$ (IQR $26 - 72\,\mu s$, $n = 10^6$) per affected *system call* without caching, matching or surpassing similar, non-enclave-based, systems.

Chapter 2

Background

2.1 Information Flow Control

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

Formally, IFC defines and enforces a non-interference policy between abstract *security contexts*. A simple 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*. [?] This relationship can generally be represented as a partial ordering over *security contexts*, formulated as a lattice. [?]

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

2.1.1 Motivation, History, and *Decentralised IFC*

IFC has recently grown in popularity as a powerful methodology for ensuring granular privacy whilst not unduly restricting access to sensitive information. IFC annotates data records with opaque *labels* referring to their confidentiality or integrity status. Rather than simply restricting access to sensitive data, as an access control mechanism would, IFC tracks data as it propagates — if an entity attempts to move data into an unknown, untrusted, or conflicting *security context* the IFC system prohibits this to prevent improper release.

IFC originated in the mid-1970s [**?**] but has not yet seen mainstream adoption. This may be because early schemes were designed around the *Multilevel Security (MLS)* doctrine set out in the *Orange Book*: [**?**] 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 has advocated *decentralised information flow control* (DIFC), introduced by Myers and Liskov. [**?**, **?**, **?**] DIFC is more granular than 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*. Tags are themselves meaningless, but used as an abstract identifier for 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.

Notation	Explanation
$A \to B$	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.

Walkthrough — Secrecy Enforcement In a typical environment, a user can only convince themselves that a text editor is safe to use if they, or someone they trust, audits the program's source code. Using DIFC, however, one can reason that a system cannot leak data without permission if the following holds;

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 before an operation is executed by the OS. Contrasting solutions handle this process differently — *Flume*, [**?**] implements a full system interposition layer, forcing all *syscalls* to pass through its userspace *reference monitor* before reaching the OS, whereas *CamFlow* [**?**] embeds its *reference monitor* in the kernel itself. In all schemes, however, this trusted component is responsible for both policy and enforcement. This project focusses on this implementation.

2.1.3 Modelling

In centralised IFC schemes, the *reference monitor* is the only entity capable of creating, changing, and assigning tags. However DIFC gives *all* processes the ability to

create and modify tags for entities they own; thus, they alone have the right to declassify them.

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

Enforcing Safe Flows (α), below, describes the conditions in which a flow can be considered *safe*. The recipient must be *at least as privileged* as the originator and cannot accept information graded below its own integrity status. Here $\preceq$ denotes any applicable preorder relation; this context uses inclusion ($\subseteq$). If a flow is *impermissible* it is denoted as $A \nrightarrow B$.

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

Information produced within a *security context* may only flow within the same context or a related *subcontext*.

Entity Creation (β) shows correct initialisation of a new object's *security context*. Logically it must be held at the same level as the environment creating it. For example, when a process creates a new file, this must 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 responsible for policies governing their assets. Therefore, a process's labelling must be dynamic. Generally, entities are 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 may modify their labelling iff they have the capability to add or remove specific tags. The set $A_s^+ \subseteq \mathcal{T}$ lists all the tags that entity A may add to its security labelling, while $A_s^- \subseteq \mathcal{T}$ holds all the tags A may remove from its labelling. These sets are modified either during creation 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) formalised this process.

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

A notable restriction is that processes must be aware of the IFC constraints imposed on them and how to interact with the system to manage their labels.

Capability Lifecycle and Delegation As per (β), an entity automatically inherits the labelling of its creator without any capabilities ($A_s^\pm, A_i^\pm = \varnothing$), therefore requiring *capability delegation* (δ). A capability held by A, $t_x^\pm$, where $t \in A_x^\pm$, is permitted to be transferred to B to act on its behalf.

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

Delegation is vital for webservers, for example. 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]

[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.

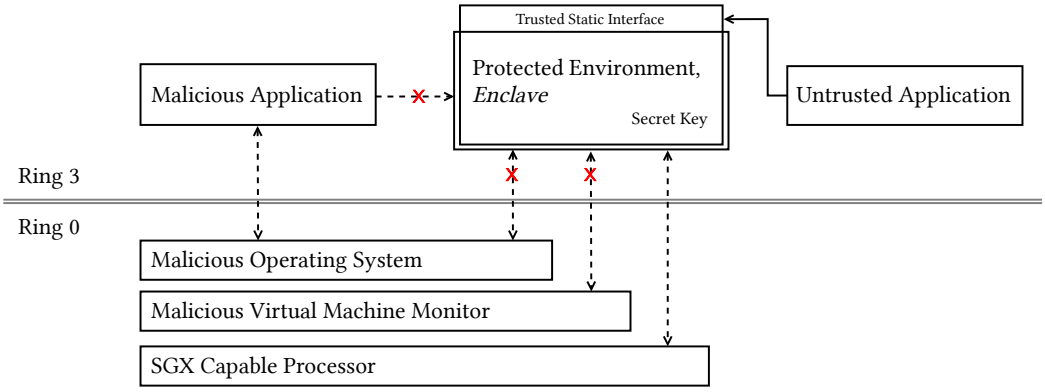

Figure 2.2: Abstract overview of SGX's protection in an adversarial environment.

2.2 Intel® SGX

Intel's Software Guard Extensions (SGX) was first detailed in 2013. [?, ?, ?, ?] Its whitepapers 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*[2] (Figure 2.2). *Enclaves* guarantee both integrity and secrecy to the application running inside, even in the presence of a malicious host.

Motivation Broadly, SGX secures sensitive applications by shielding them and their resources from any access, and to guarantee an enclave's integrity to end-users; this is achieved using attestation and measurement (see § 2.2.4). One use case [?, ?, ?] is in cloud computing, where users are forced to trust an outside party with 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, despite virtualisation. Only the provisioned CPU is able to decrypt and execute the enclave, strictly in accordance with the restrictions of the CPU platform.

2.2.1 Security Characteristics

At its heart SGX is designed to be *trustworthy*; this is achieved in several ways, including robust enclave provisioning, sealing and attestation. Intel summarises SGX's protections [?, ?] as follows;

- Memory is secured against observation and modification from outside the enclave, using an in-die *Memory Encryption Engine (MEE)*, [?] 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.
- 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: [?] mitigating this is the developer's responsibility.

2.2.2 Architecture and Implementation

The SGX platform comprises several interlocking parts (Figure 2.3), building on the core extended x86 instruction set. Information here as reported by [?, ?].

2.2.2.1 Hardware

Enclaves' state is stored securely in *Processor Reserved Memory (PRM)* a set of pages in system memory presided over by the *MEE*. *PRM* consists of two data structures; the *Enclave Page Cache (EPC)* and the *Enclave Page Cache Map (EPCM)*.

An enclave is defined by its *SECS* — generated when an enclave is created and stored in a dedicated entry in the *EPC*. A *SECS* holds important metadata including; the enclave's (system-)global identifier, its measurement hash (`MRENCLAVE`, § 2.2.4) and its memory usage.

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

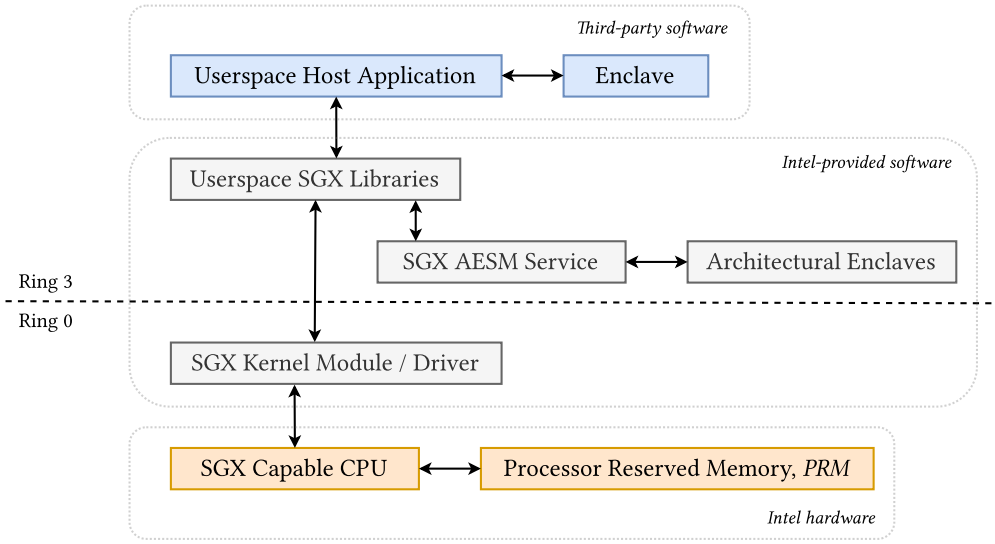

Figure 2.3: A high-level overview of the SGX hardware and software architecture.

The *EPCM* provides an index into the *EPC*: it stores access control information, ownership and validity indicators, and a marker for a page's designated use — this is not accessible from software.

PRM pages are only successfully resolved if the CPU is in enclave mode and the *EPCM* corroborates the enclave's ownership of the region; invalid requests are presented with an unused page from generic system memory. Direct Memory Access to *PRM* is always rejected.

The *EPC* is managed by the system's hypervisor or OS as typical memory, but it must use SGX-specific instructions. This is to appease the *MEE*, which is responsible for ensuring the integrity and confidentiality of this process, encrypting and decrypting pages as they cross the *PRM* boundary. An SGX driver, *isgx*, is required to allow userspace applications to use the platform and create/manage enclaves.

2.2.2.2 Userspace Services

Starting an enclave requires retrieving a *launch token* from Intel's *Launch Enclave*; this checks the enclave's validity and identity. Access to the *Launch Enclave* and other architectural enclaves is provided by the Application Enclave Services Manager

(AESM); the userspace SGX libraries facilitate communication. Other architectural enclaves include;

- The *Provisioning Enclave* — verifies the authenticity of the platform and retrieves an enclave's *attestation key* from the *Intel Provisioning Service's* servers.
- The *Quoting Enclave* — 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*.

2.2.2.3 Third-party enclaves

Enclaves 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 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 (Table 2.5[3]). The following description is illustrated in Figure 2.4.

Preparing an enclave The host application begins initiating the creation process via *isgx*, the SGX driver. *isgx* allocates the requisite number of pages to run the enclave $\langle 1 \rangle$;[4] this is tracked by the driver's internal state $\langle 2 \rangle$.

[3] A few instructions irrelevant to the explanation given here are omitted.
[4] Numbers correspond to events in Figure 2.4.

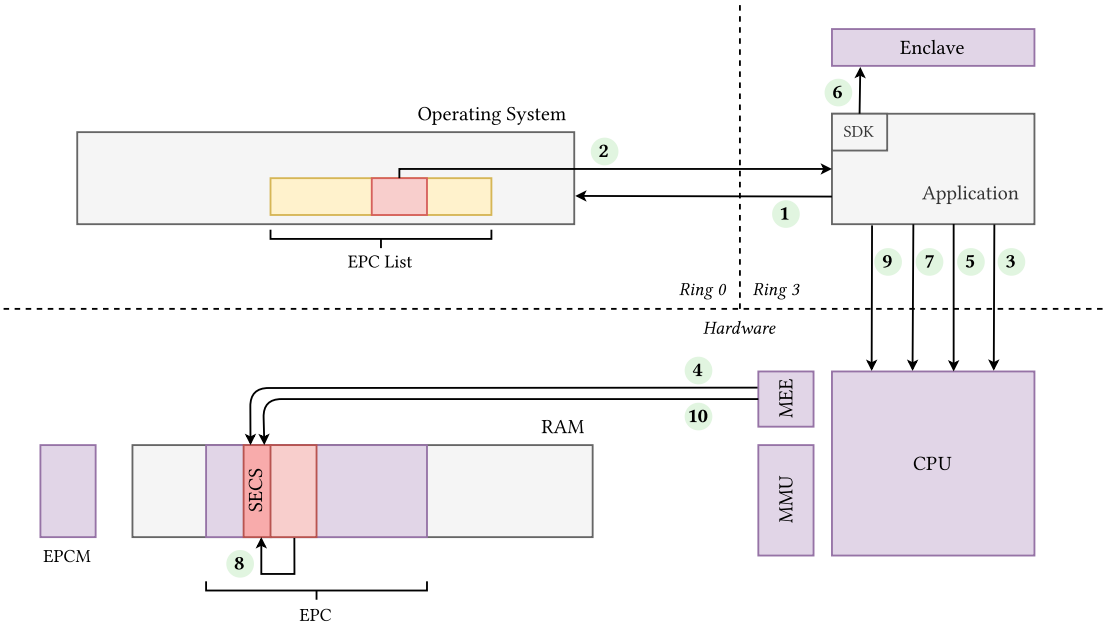

Figure 2.4: The process of creating and initialising an enclave; details given in § 2.2.3. Purple components belong to the SGX platform.

The application next calls `ECREATE` with the metadata of the enclave to be loaded ⟨3⟩; the *MEE* checks that the pages being claimed are vacant and populates the *SECS* page ⟨4⟩. Once complete the application prepares the remaining *EPC* pages using `EADD` ⟨5⟩ and loads the enclave's code and data ⟨6⟩.

Next the enclave is measured — `EEXTEND` is called ⟨7⟩, triggering the *MEE* to update the measurement hash in the *SECS* to mirror the current state of the enclave's memory ⟨8⟩. After, the *EPC* memory is finalised using `EINIT` ⟨9⟩: this operation requires the application to retrieve the `EINITTOKEN` from the *Launch Enclave*, locking the execution of the measured enclave to currently assigned processor core. Notably, pages cannot be added after `EINIT`,[5] and an enclave cannot be attested to or entered before it. Lastly, the *SECS* is updated with the enclave's final hash ⟨10⟩.

Stepping into the enclave After creation, an enclave may be invoked using the `EENTER` instruction; this can only jump to points explicitly defined in the enclave's

[5]Only strictly true in SGX v1, see § 2.2.6.

interface definition, and switches the CPU core to enclave mode. SGX uses a flag in the CPU core's *Thread Control Block* to prevent any other logical core following the current one into the enclave.

Interrupts and exceptions can be served to the enclave, just as with any other application. Control, however, is not immediately passed over to the defined handler. Instead the enclave's current state is saved and cleared to prevent leaking. The *Asynchronous Enclave Exit* routine is then invoked and enclave mode disabled. Execution post-interruption is restarted with `ERESUME`. Once execution completes, all 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 restriction of the SGX architecture is that enclaves cannot be entered from *ring 0*; [**?**] the required instructions are simply not available. Thus all host applications must run in userspace, making interoperation with the kernel challenging, as discussed in § 4.2.

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 this by creating two *signing identifiers* per enclave; `MRENCLAVE` and `MRSIGNER`. [**?**, **?**]

`MRENCLAVE` acts an a unique identifier for enclave's contents. 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

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`	Update 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 2.5: Overview of notable SGX x86 instructions in an enclave's lifecycle. [?]

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*.

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) for confidential communication between them.

Remote Attestation The Intel specification also enables 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. Using an *Enhanced Privacy Identifier (EPID)* [**?**] allows this process to complete anonymously, relying on information encoded in the CPU during manufacturing. The *Provisioning Enclave* assists in this process, especially as production enclaves are required to attest with Intel's provisioning service [**?**] before executing. Remote attestation is not explicitly required in this project, hence will not be covered further.

2.2.5 Sealing

Sealing is the encryption of data using a key unique to the generating 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 allowing state to persist through enclave upgrades.

2.2.6 SGX Versions

There are currently two versions of SGX available — 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: dynamic memory management, eased production enclave restrictions ('Flexible Launch Control'), increased PRM size support, and support for virtualisation.

2.3 Aspects of the *Linux* Kernel

Linux needs little introduction. 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 provides a brief overview of a small subset of them to support the design presented in § 4.

2.3.1 Virtual File System

Linux represents almost every component as a file, including sockets, terminals, and driver interfaces. The VFS is the transparent layer that provides this abstraction, 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. An important marker held in the superblock is it's *magic value*; this predefined code[6] indicates the underlying implementation the filesystem belongs to.

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, 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 its path in the VFS. This glue layer is responsible for building the tangible folder structure, and as the name suggests, metadata caching. A *file* consists of a *dentry-inode* pair.

File descriptors Whenever a process opens a file, it is presented with a *file descriptor* by the kernel. This structure is unique to a process, providing the gateway between

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

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 must be in the form of a null-terminated string, however they are optional and may be left empty if the attribute is just a flag. *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).

2.3.1.2 *sysfs*

sysfs is a pseudo-filesystem provided by the VFS, that the kernel uses to export its subsystems via a virtual file interface.[7] Specific instantiations of *sysfs* are used for various purposes — for example, LSMs and *SecurityFS*.[8]

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 the 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 as appropriate. Decisions taken within an LSM affect all aspects of a

[7] `/sys`
[8] `/sys/kernel/security`

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 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)* protects *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, for reference, as `security.evm`, and
2. A digital signature of this value is stored alongside using a key that is sealed either using a *Trusted Platform Module* [**?**] or passphrase.

Chapter 3

Related Work

3.1 *Flume* and *CamFlow*

Both *Flume* [**?**] and *CamFlow* [**?**] present practical DIFC systems for generic, OS-level protection in Linux. Their models are not dissimilar, with *CamFlow* refining the basic *Flume* approach. A detailed overview of the *CamFlow* model has already been presented in § 2.1, but there are important differences in the implementation of the two works.

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 Interprocess Communication (IPC) to the reference monitor, which enforces 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 In contrast, the *CamFlow* core IFC implementation lies entirely within its LSM, efficiently exploiting kernel functionality to minimise the overhead it creates. File operations in microbenchmark tests produce an 11% average overhead.

3.1.1 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. [**?**] 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. [**?**]) 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. [**?**]) 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. [**?**]), 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. [**?**]) takes a similar approach to *Flume*, using an LSM for policy enforcement, but extends it with customisation to the *Java Virtual Machine (JVM)* [**?**] to support thread-level isolation and heap-object protection. This approach has proved powerful in applying DIFC to popular processing systems such as *MapReduce* [**?**] and *Hadoop*. [**?**]

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 six years.[1] 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 when integrating it.

[1] The *linux-sgx* patch-set has seen 32 revisions; `https://lore.kernel.org/linux-sgx/`.

The *TresorSGX* [?] 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 performing at 1% the bandwidth of its kernel-embedded counterpart. The most prevalent performance hit came from communication overhead, made worse by the need to exit and re-enter ring-0.

Various other studies touch upon these issues, including:

- *Custos* (Paccagnella et al. [?]); 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 $2 - 7\%$.
- *DelegaTEE* (Matetic et al. [?]); credential delegation between two computer systems by enforcing either centrally brokered or peer-to-peer *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. [?]); 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. Performance overheads are 100%.

3.3 Dataflow Protection using SGX

Research into applying SGX 's protections to large-scale distributed computation has accelerated recently — the most prominent projects are detailed here.

- *SCONE* [?] presents a secure container framework for *Docker*. [?] Using a secured version of the standard library for C it transparently encrypts and decrypts I/O crossing the container's boundary. The authors claim $\times 0.6 - 1.2$ the performance of native throughput.
- *VC3* [?] secures *Hadoop MapReduce* computations — the *Hadoop* platform is not considered part of the TCB, thus allowing the system's security invariants to re-

main unaffected if it were to be compromised. The reported performance overhead is 8% (for full read/write integrity).

- The *Maru* project [**?**] 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 was porting the *JVM* to function efficiently inside an enclave; SGX v1 restricts the EPC size to 128MB, severely penalising applications that struggle to run in relatively small memory footprints.
- *Ryoan* [**?**] 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* [**?**] for the host application.

Chapter 4

Citadel

In this chapter we introduce a prototype implementation of a modular, SGX-protected *reference monitor* — Citadel. We start by considering this project's motivation and discussing the challenges faced. Then, we explain the three-part architecture, relating design decisions to its DIFC model. We discuss the architecture's performance and effectiveness in § 5.

4.1 Motivation

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

Fundamental Properties of a Reference Monitor

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

with, ideally, a restricted TCB.

- *Tamper proof.* The integrity of a reference monitor, which presides over the system's authorisation mechanisms, 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 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] Therefore we must question whether Linux alone can provide a reference monitor implementation the guarantees it requires, [?, ?] thus motivating the use of SGX.

Applying SGX to this problem brings two attractive benefits;

- The system's IFC policy can be evaluated both during offline analysis and online using *attestation*, building other enclaves' confidence in the underlying system.
- SGX's hardware protections are very capable of 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 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 significant, 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`

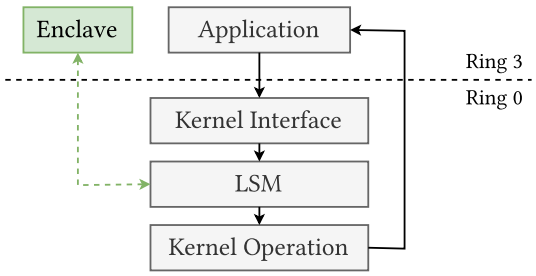

Figure 4.1: Abstract *syscall* control flow route. Grey components show the natural Linux design. Green additions highlight the externalised enclave LSM component.

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, we seek to minimise the overhead caused by its integration, while maintaining *safety* (every operation must be mediated). This situation is reminiscent of *exokernels*; [?] forcing a portion of the system into userspace could, if treated carefully, help overall performance. [?]

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 until needed. Backflow of information is facilitated asynchronously, but no additional kernel relay is required.

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

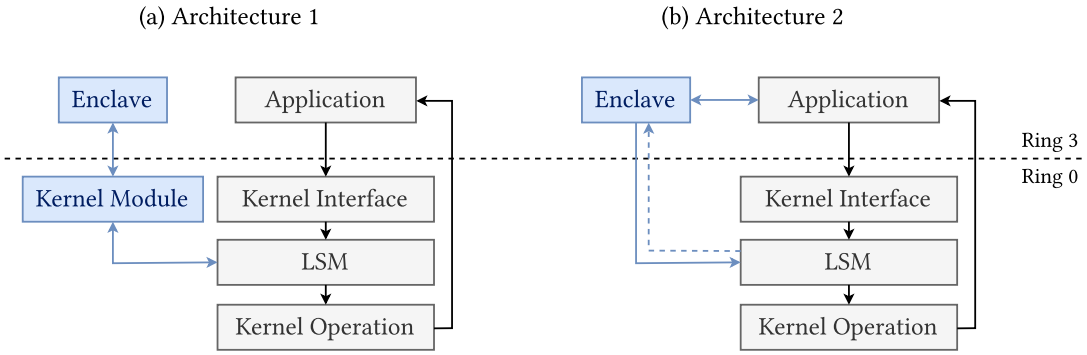

Figure 4.2: Two possible enclave integration designs.

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 an extension because *policy decisions and enforcement are no longer one and the same.*

Preliminary experiments showed that the performances of the two architectures were similar in light workloads, but that *Architecture 1* degrades significantly with resource contention. Additionally, as will be explained in § 5.2.1, 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 is not privy to internal kernel datastructures such as `task_struct`, which will store the taint and capabilities of processes. A potential solution would be 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. [?]

As a final comment, it must be noted that SGX is not flawless; § 5.2.3 discusses the impact of this 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 directly extends the one presented in § 2.1.3.

4.3.1 Reservations

Previously we defined the concept of a *safe flow*, $A \to B$, which underpins our IFC restrictions. In previous works permission 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.

$$\textit{operation} \to \boxed{\textit{reference monitor}} \xrightarrow{\textit{decision}} \{0, 1\} \tag{4.1}$$

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

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

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

Further, we define a shorthand, t^{α};

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

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

For a process A, $A_r \subseteq \mathcal{R}$ defines 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 *validity* with an indicator function, $\mathcal{V} : \mathcal{R} \mapsto \{0, 1\}$. A reservation can only be used if valid; invalid reservations are discarded.

4.3.2 Permissible Operations

Satisfiability To determine if an operation is permissible, the *constraint reservation* representing it is compared against reservations held by the process. For example, $t^{\{\texttt{file\_read}\}}$ is the constraint for reading a file tagged 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*, $A \xrightarrow{\omega} t$; process A may perform operations ω on an entity tagged t. An operation is only permissible if the process holds a reservation explicitly granting permission (4.4).

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

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:

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

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

Together, these define the relationship between an abstract policy space ($A \to B$, § 4.4.2) and concrete implementation ($A \xrightarrow{\omega} \tau$, § 4.4.1). A policy decision may grant

a greater set of permissions than asked for (4.6) — e.g. allowing both read and write when only write was explicitly requested. [?, ?]

Some 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 entity; they are used to model Linux functions such as `pipe()` and unclaimed tainted files.

To facilitate this, all processes are assigned a unique tag $p \in \mathcal{T}$, and any files it creates are 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 expression for a *permissible operation* now becomes;

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

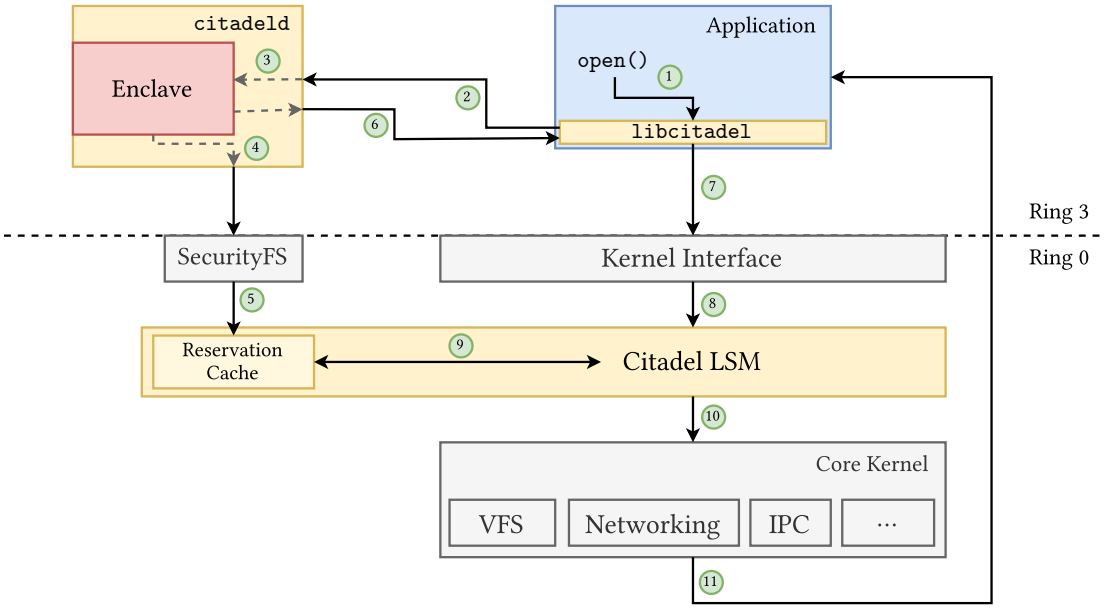

Figure 4.3: High level overview of the Citadel architecture.

4.4 Implementation

Citadel consists of its LSM, `citadeld`, and `libcitadel`. Each plays an essential, symbiotic role in the reference monitor's operation. The prototype is over 9,000 lines of C and C++, extending the Linux kernel build system (§ 4.4.6). This section presents Citadel's architecture, guided by Figure 4.3.

Analogy The system is well illustrated by the *will-call* system used by theatres — customers (*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 comprises 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. Policy components need not be aware of the enforcement strategy to successfully express their protection schemes. Inter-domain communication is discussed in § 4.4.3.

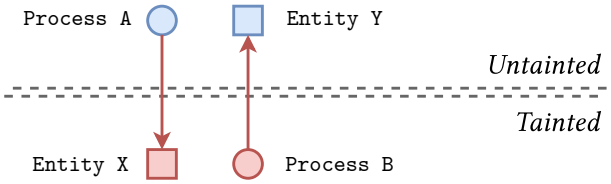

Figure 4.4: Accesses across the taint boundary taint the untainted party.

4.4.1 Enforcement

The CITADEL LSM tracks all entities within the Linux system by 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 and additive — entities are only policed if involved in a successful operation crossing the taint boundary (Figure 4.4). In additional to automatic propagation, the policy domain may amend the taint for most entities.

The following metadata is tracked for each entity:

— **Active**. Processes only — these require many 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*. Non inode-backed tracking is discussed in § 4.4.2.5.

4.4.1.1 Identifiers

Entities are tagged with a single, randomly-generated 128-bit identifier, A security policy may maintain internal pseudonyms for secrecy and integrity, but must convert 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*. These occupy the `security.citadel` namespace, and ensure that taints and identifiers persist between boots. Certain entities may be *anonymous*, indicated by an anonymous flag, if

their identifier is not present as an *xattr*, either because the entity does not support *xattr* or those with temporary identifier (§ 4.4.1.5).

4.4.1.3 Permissions

Tainted processes must hold a valid reservation to perform any operation allowing data to flow to another entity — enforcement strictly follows the rules in § 4.3.2. Untainted processes bypass all checks, and thus lie outside the IFC model; security implications are discussed in § 5.2.2

4.4.1.4 Reservation Cache

When the system's policy enclave presents a new reservation to the LSM, it is stored in the *reservation cache*. Implemented as a red-black tree, this 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 updates the process's reservation list by;

1. *Installing pending tickets.* All reservations are moved to its internal reservation list, ready for inspection.
2. *Disposing of expired entries.* 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

Every newly spawned process is privately tagged by the LSM as if it were a passive entity (§ 4.3.3). The purpose of this identifier is to enable association any private, passive entities it creates. This includes the file descriptors provided by `pipe()`, and any new files it creates using `open()`. Processes always have permission to access their transient entities, and external entities can only gain access rights if they;

1. Are a child processes requesting access to their parent, or

2. The process officially *claims* them via the policy enclave, giving them an independent tag and removing 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 without the same rights as their parents — open file descriptors will not function without revalidation (§ 4.4.4), and children must request the right to their parent's transient entities to use pipes or similar. It is the policy enclave's responsibility to validate that the security contexts of the parent and child have not diverged.

4.4.2 Policy Components

`citadeld` represents the policy counterpart to the LSM's enforcement, including 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 is presented with a simple, event-driven interface, streamlining their implementation, and allowing more emphasis to be put on 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 that any operation is 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 ($A \rightarrow *$).

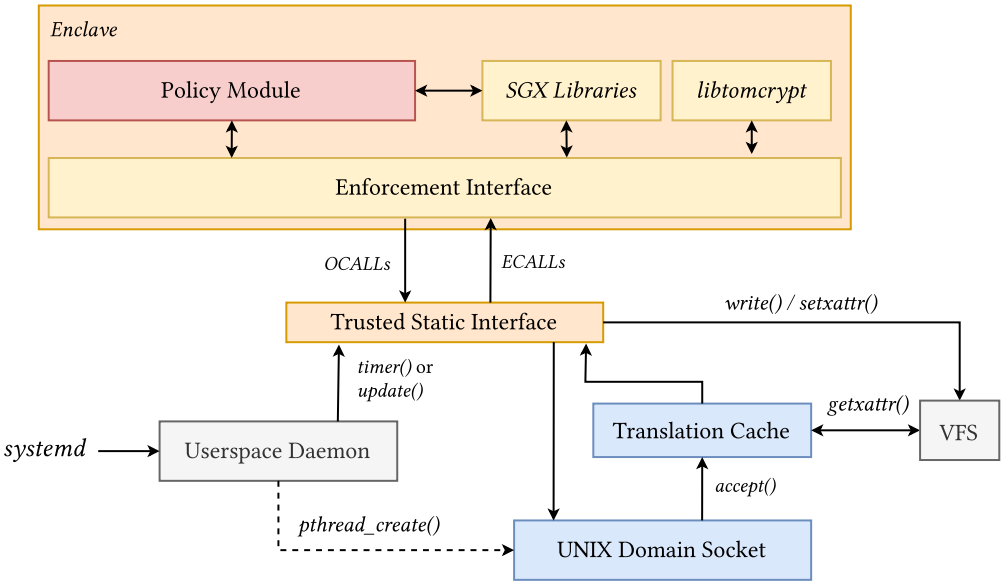

Figure 4.5: Overview of the components inside `citadeld`.

Operations Entity operations, Ω, are presented as `citadel_operation_t`, a simple bit mask over the operations Citadel recognises. Similarly, policy decisions are represented using `citadel_response_t`; these may be *approved*, *rejected*, *error*, *invalid*, *granted*,[5] and *forged*.[6]

4.4.2.2 Host Application

There are several steps before presenting requests to the resident policy module (§ 4.4.3). Requests often refer to absolute filepaths, requiring retrieval of their tags, if they exist — security *xattrs* serve these requests. Translation is performed preemptively depending on the operation requested, and results are cached in a *translation cache* to minimise overhead. This is implemented using *sparsehash*,[7] and great care is taken to detect stale entities that may confuse the internal decision process.

[5] Approved, and confirming that the process is recognised as the owner of the entity.
[6] See § 4.4.5.
[7] `https://github.com/sparsehash/sparsehash`

4.4.2.3 Enforcement Interface

The policy module is interchangeable, but the enforcement interface acts as the enclave's backbone. All requests are routed through it to detect forgery or invalid data, and all information leaving it is formatted and signed[8] as appropriate. The process of installing reservations created by the enforcement interface is detailed in § 4.4.3.

4.4.2.4 `libtomcrypt`

We ported *libtomcrypt*,[9] a leading open-source cryptography library, to function inside an SGX enclave as `citadeld` requires many encryption mechanisms on top of those provided by SGX. Porting was achieved by replacing *libtomcrypt*'s backing precision arithmetic library with an SGX-aware version of *GMP*,[10] 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.

4.4.2.5 Shared Memory

Citadel also supports the tagging and restriction of shared memory (SHM). This is managed directly using *System V* identifiers granted to allocated memory segments instead of inodes. Internally, restrictions function as files do, but per-access mediation is not directly possible – we can only detect when segments are allocated and attached. Thus this workflow requires special consideration and a new reporting mechanism from the LSM back to the policy enclave. Using a specialised *xattr* interface (4.10) to drive a request-response model, the LSM tracks and reports the PIDs of everything that has touched an SHM segment.

$$\texttt{security.citadel.shm.[shm\_id]} \longrightarrow \{145, 267, 1120, ...\} \tag{4.10}$$

[8] Encryption is discussed in § 4.4.3

[9] `https://github.com/libtom/libtomcrypt`

[10] `https://github.com/intel/sgx-gmp`

4.4.3 Communication Pathways

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

1. Applications $\longleftrightarrow$ 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 can communicate with the reference monitor, a special tag, $\tau = 2^{128} - 1$, is assigned, asserting the reference monitor's ownership of it and whitelisting it in the LSM.

2. Policy Enclave $\longleftrightarrow$ LSM.
   Communication uses two mediums; *SecurityFS* and *xattrs*. All messages are encrypted using AES-256-GCM [**?**, **?**]; the key is chosen during initialisation (§ 4.4.3.1).
   Reservations are installed using a custom *SecurityFS* interface[11] and synchronously inserted into the reservation cache. The policy enclave may invoke an operation directly on a file using `setxattr()`, which the LSM intercepts, triggering it to enact the required changes. One common use of this is entity tagging.

3. LSM $\longrightarrow$ Applications.
   To verify their identities with the policy enclave, applications present a *ptoken* with each request (§ 4.4.5.1) which is generated by reading from a public *SecurityFS* interface.[12]
   Additionally, `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 any time afterwards, meaning that the LSM must be capable of operating independently. In this case the system will tend towards a state of complete lockdown (for tainted processes). Thus the mechanism by which the LSM and policy enclave ini-

[11] `/sys/kernel/security/citadel/update`
[12] `/sys/kernel/security/citadel/ptoken`

tialise 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 as a sealed entity; sealing here uses `MRSIGNER`, allowing any policy enclave provided by the *sealing authority* to fully function. Sealed keys are compiled into the kernel, available via *SecurityFS*.[13]

Once a policy enclave has been initialised it must verify itself; the LSM issues a random challenge[14] encrypted using the enclave's public key, and expects a reply containing the correct challege token.

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

Given E_S is only held sealed, any entity providing a valid challenge response 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 underlies all future communication.

CITADEL uses *AESNI* [**?**] to provide AES protection as all SGX-capable processors support it. It provides hardware-accelerated AES operations to achieve an encryption bandwidth over 1 Gbps, far exceeding the capacity required, thus adding negligible overhead. The system's AES key updates with every message sent from the enclave using an SGX-approved source of entropy;[15] this adds minimal overhead and constitutes good practice. A copy of the first key presented in the challenge-response is retained and used in cases when a static key is essential.

[13] `/sys/kernel/security/citadel/sealed_keys`
[14] `/sys/kernel/security/citadel/challenge`
[15] $new \leftarrow old \oplus update$

4.4.4 `libcitadel`

CITADEL provides a userspace auxiliary library to make integrating existing programs easy and unobtrustive. For each mediated *syscall* (e.g. `open()`) it provides a proxy function (`c_open()`), thus requiring no major changes to applications' workflows.[16] A good example of this in action is the ported version of *Nginx* (§ 5.1.4). `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 `citadeld`'s domain socket. A zero-copy approach[17] helps minimise latency on both sides; this is optimised for in the protocol design. Each communicant verifies the PID of the other party (§ 4.4.5.1).

Caching at this level has a tremendous impact on overall performance. When reading a large file a program may make thousands of calls to `c_read()` — always calling to `citadeld` would be wasteful, as processes usually have enough information to infer their current position.

Therefore every process maintains a list of *expectations* — the reservations it believes it has, including their validity — and inferred taint status. They cannot precisely know the true values, especially as the policy enclave may grant different permissions than asked for, but in *Nginx* over 97% 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 involves manual work, including fetching its *xattr* tag with `fgetxattr()` and estimating the expiration time of the LSM's underlying reservation. `libcitadel`

[16] Future work would integrate this directly into `libc`.
[17] Excluding copying in the kernel and transferring the request into the enclave.

```
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 4.6: The `libcitadel` shim function for `open()`.

attempts to revalidate tickets before expiry to avoid unexpected drops in service; this is particularly important for applications unaware of Citadel.

Special care is required with child processes. Children are given a copy of their parent process's state, including its `libcitadel` cache. Although the LSM passes no reservations 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 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 and prove identity.

4.4.5.1 *ptokens*

Before a process may interact with `citadeld`, it must retrieve its *ptoken* from the LSM.[18] The purpose of this (4.11) 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 encrypted using K, the system's designated static AES key, which is unknown to the process.

$$ptoken \rightarrow (\text{citadel\_pid}, \text{identifier}, \text{token}, \{\text{identifier}, \text{token}, \text{pid}\}_K) \tag{4.11}$$

Whenever a process connects to the `citadeld` socket, its identity is retrieved from the underlying transport mechanism (Listing 4.7). At both the sender and receiver the identity of the other is verified using this method, and additionally `libcitadel`

[18] 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;
```

Listing 4.7: PID retrieval from an active domain socket.

expects the decrypted *token* to be returned by `citadeld`, inspiring confidence that the response is legitimate.

4.4.5.2 PID Protection

The LSM also watches for PID forgery, as it is possible in certain scenarios for PIDs to be modified with the help of a malicious kernel module (Appendix A). This would be detrimental for the LSM's integrity, allowing a process to silently assume another's identity. Therefore the LSM stores a process's PID within its security structure and routinely checks to ensure it does not change unexpectedly.[19] 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 system's *Sealing Authority*; without this operation will fail. A preparatory script achieves this by;

1. Downloading the kernel's source and inserting the Citadel LSM.
2. Generating two *OpenSSL*[20] 2048-bit RSA keys in *DER* format — the kernel's *Crypto API* requires keys to present themselves as *ASN.1 structures*.[21]
3. Compiling and launching Citadel's *preparatory enclave*, 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.

[19] A valid change would be on `fork()`, in which case the stored PID should equal the parent process's.
[20] `https://www.openssl.org/`
[21] `https://tls.mbed.org/kb/cryptography/asn1-key-structures-in-der-and-pem`

4. Generating an interface file (`keys.h`) 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 deleted.
5. Building and signing the policy enclaves.
6. The kernel may now be compiled.

Chapter 5

Evaluation

Three core questions hang over CITADEL's viability — its security, applicability, and performance. This chapter presents a thorough investigation of the prototype's performance and a discussion of its security implications and application to real-world scenarios.

5.1 Performance

The CITADEL prototype demonstrates impressive performance, matching, and in places surpassing, related approaches, despite its the architectural disadvantage. We present its behaviour relative to native Linux kernel as follows;

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 following results are best compared to *Flume* [**?**] — *CamFlow*, although implemented similarly, has a different scope that this project. *Flume* reports a 40% decrease in real-world performance; we report $\sim 25\%$ (§ 5.1.4).

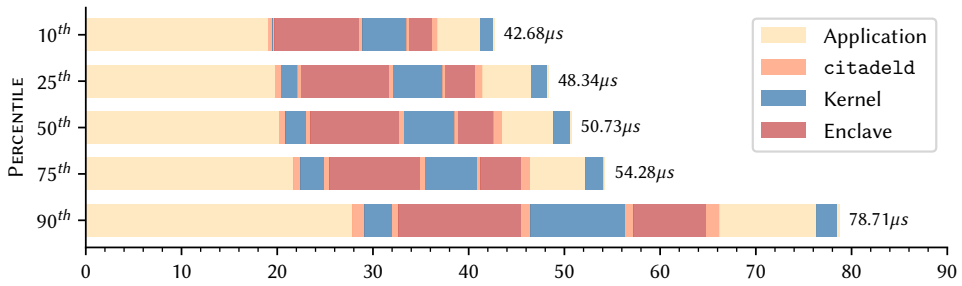

Figure 5.1: Control flow inhabitation for `libcitadel`'s `c_open()` function, $n = 100$.

5.1.1 Evaluation Environment

The research machine used for evaluation contained a quad-core Intel® Core™ i5-6600 (supporting 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 with the *transition_using_threads* optimisaton. [?] Linux v5.6.0 was used as the base kernel.

Both Tables 5.2 and 5.6 report the sample mean and standard deviation. Figures 5.3, 5.4, and 5.5 plot the sample medians and interquartile range (IQR) for each point. The Wilcoxon paired signed rank test was chosen to determine statistical significance at 5% confidence. [?]

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 5.2 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 column, *Cache Miss*, gives the overhead when caching is disabled, thus including communication with `citadeld`.

[1] Reported by the `dd` tool.

	Native	Citadel		
		Amortised	*Cache Miss*	99^{th} *%ile*
`open()`	1.675 ± 0.076	6.083 ± 0.129	50.133 ± 1.482	$2.38\times$
`read()`	5.724 ± 0.206	7.010 ± 0.192	54.736 ± 1.556	$1.26\times$
`write()`	14.340 ± 0.208	15.597 ± 0.250	63.824 ± 1.902	$1.05\times$
`close()`	0.651 ± 0.005	0.718 ± 0.011		$1.10\times$
`socket()`	1.446 ± 0.179	3.156 ± 0.291		$1.02\times$
`bind()`	0.762 ± 0.023	1.911 ± 0.183	49.110 ± 1.746	$2.78\times$
`listen()`	0.705 ± 0.015	1.882 ± 0.149	48.411 ± 1.386	$2.91\times$
`connect()`	16.570 ± 0.278	17.961 ± 0.330	66.273 ± 2.147	$1.05\times$
`shmget()`	1.880 ± 0.122	1.913 ± 0.111	49.326 ± 1.466	$0.98\times$
`shmat()`	0.420 ± 0.005	1.575 ± 0.134	47.997 ± 1.560	$0.99\times$
`shmctl()`	0.418 ± 0.005	0.743 ± 0.083	45.912 ± 1.114	$0.97\times$
`shmdt()`	0.415 ± 0.003	1.342 ± 0.040		$1.01\times$
`pipe()`	1.110 ± 0.061	1.288 ± 0.069	47.334 ± 1.147	$1.02\times$
`mkfifo()`	3.865 ± 0.048	11.509 ± 0.405	59.623 ± 1.788	$1.93\times$
`fork()`	47.866 ± 3.175	48.647 ± 3.457	81.174 ± 3.829	$15.77\times$
`citadel_init()`	$-$	0.801 ± 0.009	34.940 ± 1.329	$-$

Table 5.2: `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. The difference between Citadel and Native Linux at the 99^{th} percentile is also presented. $n = 10^6$.

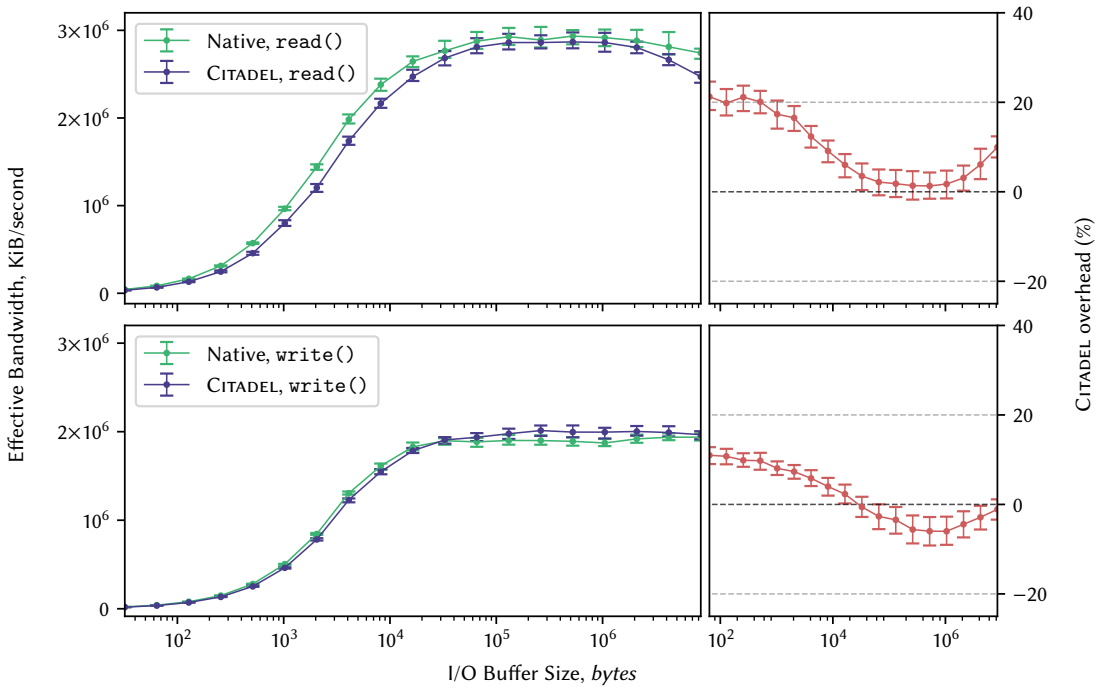

Figure 5.3: Effective `read()/write()` bandwidths for both the native Linux kernel and CITADEL. The percentage overhead is also presented. $n = 200$ per buffer size.

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` (median $26\mu s$);[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 `citadeld` cache. During these experiments, `citadeld` was shown to reliably handle over 30,000 requests/second and between $90 - 100\%$ CPU usage of a single thread.

[2]Included in the *Application* regions.

Figure 5.3 plots observed effective bandwidth whilst reading from and writing to a 16 MiB file with different sized buffers; the corresponding percentage overhead inflicted by Citadel is plotted to the right. The benchmark driving this was adapted for Linux from one written by R. Watson for FreeBSD. [?] The results clearly show Citadel having a more adverse effect on performance for smaller buffer sizes; unsurprising, as smaller buffers force a larger number of calls to `read()` / `write()`. It is unclear why Citadel provides better performance for large buffers with `write()`, an unexpected artefact — the difference is statistically significant for buffers in the range 256KiB and 2 MiB, and reproducible. More work is required to determine the root causes, but hypotheses include the slight optimisation afforded by regular, small delays easing pressure on microarchitectural caches.

5.1.3 IPC Microbenchmarks

Again using the modified Watson benchmark, we investigated Citadel's effect on end-to-end IPC performance. We investigate *pipes*, *local sockets*,[3] and *regular sockets*, between 2 threads (Figure 5.4) and between a parent and child process (Figure 5.5).

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 appears that Citadel affects the performance between 2 threads slightly more than 2 processes, but in fact Citadel performs near-identically in both. Native performance is more heavily optimised when sending between two threads; Citadel overshadows any latency gained by the kernel.

In a similar way to `write()`, Citadel unexpectedly outperforms the native kernel in both contexts using `pipe()`. The readings are noisy, but statistically significant for buffers in the range 16 KiB to 8 MiB, and reproducible. The cause is again unknown, but observing that the native kernel's throughput halves after buffers of 8 KiB, we suspect that this is the result of cache exhaustion or inopportune paging. Notably, Citadel's readings exhibit a far larger IQR for large buffer sizes than the native kernel, a side effect that is repeated in real-world testing.

[3] `socketpair()`

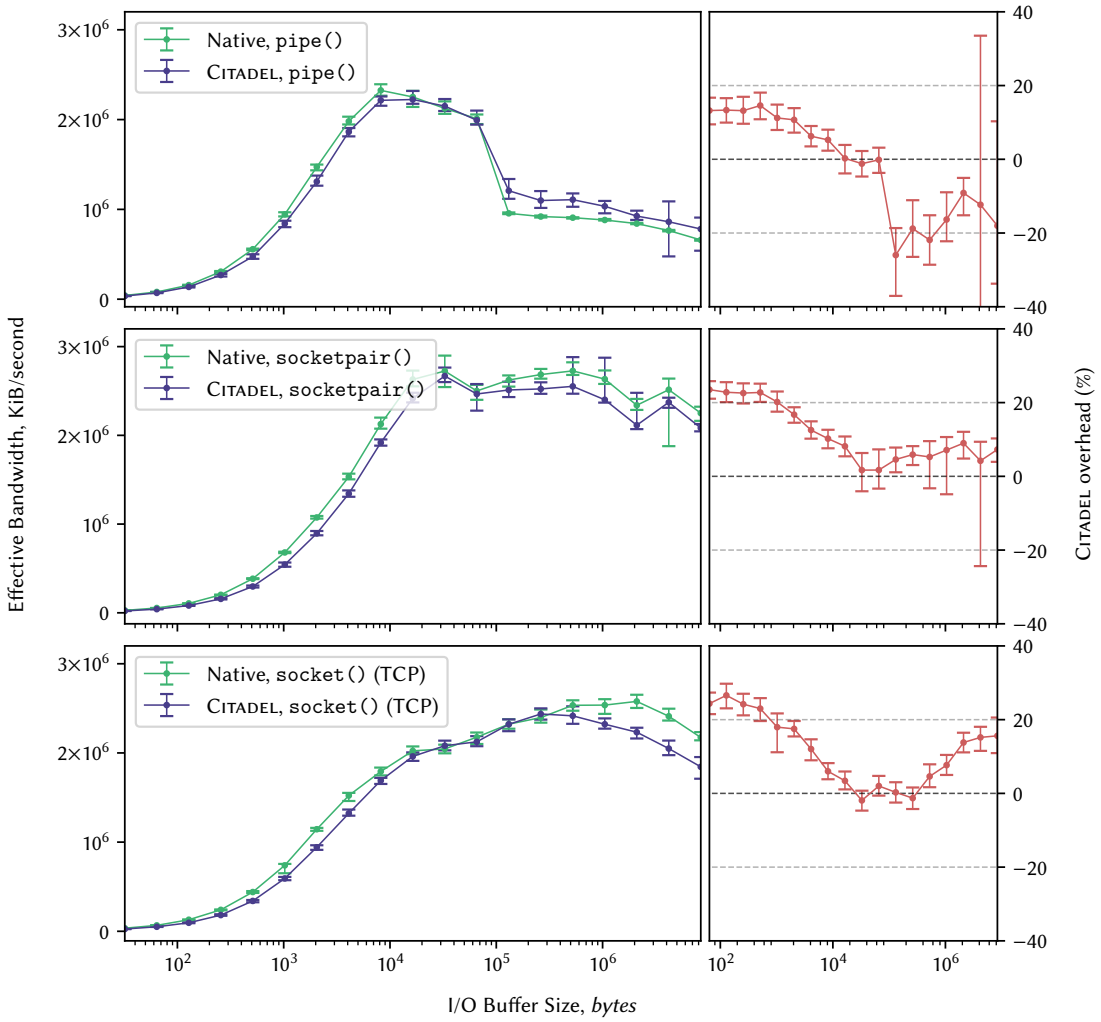

Figure 5.4: Effective bandwidths for various types of IPC between *2 threads*, $n = 200$.

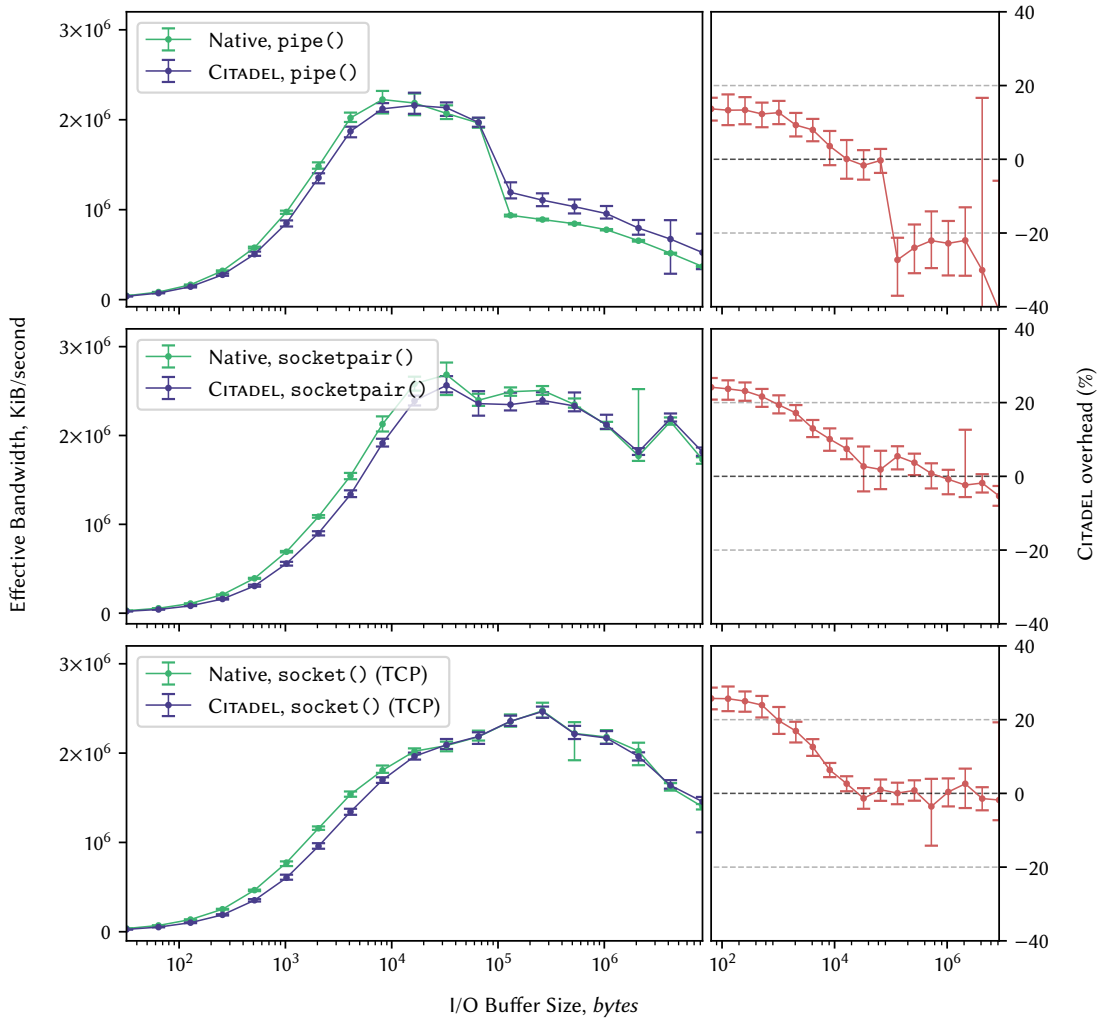

Figure 5.5: Effective bandwidths for various types of IPC between *2 processes*, $n = 200$.

	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 5.6: 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[4] to function alongside Citadel. No optimisations were made to the codebase — the only changes made replaced core `libc` function calls to use their `c_*` `libcitadel` counterparts.

Two trials were run; a low-latency benchmark[5] 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 set up to use the *loopback* interface[6] to eliminate any interference from outside the OS.

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

The results are not surprising (Table 5.6). 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 to the native kernel, only differing by its larger sample standard deviation. This trial is also interesting as this is the first time we see file descriptor revalidation happening automatically on `read()` and `write()`. The CPU overhead from `citadeld` was $< 1\%$.

5.2 Security

5.2.1 Citadel TCB

One of the core initial goals of this project was to build an enclave-based reference monitor whilst *minimising inflation to the system's TCB*; to this end we present the trusted components of a Citadel system.

- The SGX Platform, including all libraries and *isgx*.
- The *policy enclave* implementation.
- The userspace `citadeld` application; discussed further below.
- The core Linux kernel, including the Citadel LSM and the Linux VFS.
- The Intel *AESNI* Linux driver.

We also assume that the build environment is entirely trusted. A notable exclusion from the TCB is the majority of Linux drivers and other kernel modules — this was a strong motivation for using SGX, as it can effectively defend against malicious and misbehaving ring-0 parties.

However, how might the system defend against `citadeld`, an unprivileged, user-space application, being replaced by a malicious adversary?

`citadeld` may be protected in exactly the same way that Citadel defends its socket, `/run/citadel.socket`, with a reserved identifier. For example, opting to mark it with an *xattr* such as `security.citadel.daemon` with a *nonce* value during the build

process provides the LSM ample confidence that an executable is a valid product of a CITADEL build,[7] and protects it from tampering. This may also be valuable for defending the enclave object files themselves; although they cannot be tampered with, an adversary may try to remove them to deny service. These features would require extending the CITADEL build system further.

Secondly, can the kernel actually be held to the same integrity level as an enclave? How trustable is a system that enclaves can only attest to half of?

Enclaves are unique in that their online attestation process inspires absolute trust, but offline provenance can be equally valuable. Although not SGX-based, *UEFI Secure Boot* [**?**] is an industry-standard mechanism for verifying whether an OS is legitimate. Ensuring that a CITADEL system's installation is trustworthy by having it properly signed by a trusted party defends against many types of attack, and using IMA (§ 2.3.2.1) strengths it further. Assuming that an OS installation is verified, trust in CITADEL revolves around the policy enclave — if it believes the system is legitimate then other enclaves can as well. It alone has the power to decide permissions, giving weight to its endorsement of its own TCB. This, however, is undermined when running on a hypervisor, as the LSM's integrity may be compromised.

Thirdly, does the kernel protects itself adequately from malicious kernel modules?

Effective protection is possible if carefully executed. Appendix A presents a proof of concept kernel module that changes a process's PID dynamically. This is highly concerning, as a process's PID is its core identifier in CITADEL's eyes; defence was discussed in § 4.4.5.2. The Linux kernel is a soup of *exported* and *unexported symbols*,[8] which is exploitable to access internal functions never designed to be called from a different context. This work does not assess the implications for the LSM framework, but highlights the potential need for *defensive programming* when designing the LSM.

[7] Using the `bprm_check_security` LSM hook.

[8] Symbols include functions and variables held in the global namespace; exporting is the process of exposing it publicly to be called by third parties.

We believe that Citadel, with its restricted TCB, can be trustworthy, creating a system of integrity checks based upon, and complementing, the trust placed in SGX itself.

5.2.2 IFC Model Implications

The Citadel IFC model deviates from the designs of Pasquier et al. and Krohn et al. in three key ways;

1. Policy and enforcement decision are separated.
2. Passive entities may exist in a temporary *transient* state.
3. Operations between untainted entities are not mediated.

The first is handled with an extension to the core model to make policy decisions explicit and communicable (§ 4.3.2). This relationship is this work's core focus, and Citadel relies on conservative assumptions — if not explicitly granted, permission is withheld. By default the system tends towards complete lockdown, a defensive measure to preserve *safety* if subjected to denial of service.

The second point is justified with an extension to the *creation flow* rule (§ 4.3.2, (4.7)). Transient entities are only created from a secure context, under which they automatically assume the labelling of their parent entities. This may informally be considered an extension of the process's internal state, and held to the same restrictions. Such entities may only alter their security context (via another active entity) after being declared explicitly, leaving their transient states.

Regarding the final point — tainting in Citadel assumes the worst, treating any potential infraction as cause for mediation; only the policy enclave may clear a taint. Assuming all sensitive entities are correctly labelled, Citadel will protect them against anything within the system. Unlabelled entities are assumed to be in the public domain, which allows normal execution and access control to proceed until the taint boundary is crossed.

5.2.3 SGX Vulnerabilities

This work assume that the SGX platform is itself secure; a successful attack on SGX obviously compromises any protection offered by Citadel. A number of SGX vulnerabilities have been detailed; [?, ?, ?, ?, ?] these are effective and highly concerning, but mitigating them is not within this project's scope.

5.3 Use Cases

Our goals for Citadel included using it as a platform for reasoning about the relationship between an enclave and its host. Standard workflows for creating an SGX application use a library OS, such as *SGX-LKL* [?] and *Graphene-SGX*, [?] to create a synthetic Linux environment inside the enclave supporting the primary application — this approach subjects the enclave to some of the same flaws and issues as the underlying OS. Could we reach a point where the host OS is trusted to hold custody of sensitive assets, instead of a trend toward pure isolation?

Two hypothetical scenarios are presented — one requiring secrecy, and one integrity — to illustrate how Citadel could aid enclave-application development.

Scenario 1 A social-media company provides a GDPR platform to fulfil members' requests for an archive of their personal data; these may exceed 10 GB. Processing and collation happen inside an enclave, and an external service authenticates requests. Must the enclave seal archives after creation, before storage, and unseal them when requested? A solution using Citadel may offload unencrypted archives to the custody of the host OS, using IFC's secrecy mechanic to prevent unauthorised release. On request, the webserver requests permission to declassify the archive from the authentication authority — no penalty need be paid for encryption.[9]

Scenario 2 An Apache Spark application partitions input data into a large number of shards. Assuming that shard-processing is protected inside an enclave, do shards need to be cryptographically signed to verify their provenance? Citadel would entrust this

[9] Disk encryption should be used for offline protection.

tracking to the policy enclave, which, once attested, may be considered a part of the application's trusted components.

Although Citadel may not be suitable for the most sensitive processing tasks — enclaves are still vital here — it offers a lightweight protection mechanism that could potentially be, in the interest of performance, trusted in a supporting role.

Chapter 6

Conclusions

This dissertation presented Citadel, a modular, enclave-backed reference monitor which securely and verifiably implements IFC methods in the Linux kernel. By separating policy decisions and enforcement, we demonstrated a feasible approach to deep kernel integration using Intel SGX; the prototype leverages Linux's security framework to realise decisions at the lowest level of the OS. Citadel optimises for performance via an auxiliary library, which conservatively predicts a process's security context, enabling unobtrusive application integration.

A full implementation of the Nginx webserver running on Citadel validates this work using real-world performance benchmarks; the most punitive trial produced a 25% overhead, but other scenarios reported performance parity with the native Linux kernel. Verifying the methods presented here should be the next step, but an extension of Citadel in a distributed setting also has great potential; inter-machine attestation will likely establish an exceptional degree of trust between remote components.

Further work is required before Citadel is fully realised and production-ready, but this project successfully demonstrates the viability and potential of a symbiotic enclave-kernel relationship, which, in the long run, may prove valuable for both.

Appendix A

PID Tampering: Proof of Concept

```
1  static void* retrieve_symbol(const char *sym) {
2     static unsigned long faddr = 0;
3
4     // Compare kernel symbol with query.
5     int symcmp(void* data, const char* sym, struct module* mod,
6                unsigned long addr) {
7        if(!strcmp((char*)data, sym)) {
8           faddr = addr;
9           return 1;
10       }
11       else return 0;
12    };
13
14    kallsyms_on_each_symbol(symcmp, (void*)sym);
15    return (void*)faddr;
16 }
```

Listing A.1: Expose unexported symbols from the global namespace using `kallsyms`.

```c
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 A.2: Exploit unexported symbols to change the PID of the current process.

Acronyms

AES Advanced Encryption Standard. 37, 38, 41

AESM Application Enclave Services Manager. 10

AESNI Advanced Encryption Standard New Instructions. 38, 53

API Application Programming Interface. 42

ASN Abstract Syntax Notation. 42

CPU Central Processing Unit. 1, 8, 10, 12, 15, 52

DER Distinguished Encoding Rules. 42

DIFC Decentralised Information Flow Control. 3–6, 19, 20, 23, 30

EPC Enclave Page Cache. 9–13, 22

EPCM Enclave Page Cache Map. 9, 10

EPID Enhanced Privacy Identifier. 15

EVM Extended Verification Module. 18

GCM Galois Counter Mode. 37

GDPR General Data Protection Regulation. 56

GMP GNU Multiple Precision Arithmetic Library. 36

HDFS Hadoop Filesystem. 22

HTTP Hypertext Transfer Protocol. 52

I/O Input/Output. 20, 21, 25, 36

IFC Information Flow Control. 2–5, 7, 19, 20, 22, 24, 25, 28, 33, 55, 56, 59

IMA Integrity Measurement Architecture. 18, 54

IPC Interprocess Communication. 19, 20, 45, 49

IQR Interquartile Range. 2, 46, 49

JVM Java Virtual Machine. 20, 22

LoC Lines of Code. 24

LSM Linux Security Module. 2, 17–20, 24, 25, 31–33, 36–39, 41, 42, 46, 53, 54

MEE Memory Encryption Engine. 9–12

MLS Multilevel Security. 4

NIC Network Interface Card. 45

OS Operating System. 1, 2, 5, 10, 19–21, 52, 54, 56, 59

PID Process Identifier. 36, 38, 39, 41, 42, 54

PRM Processor Reserved Memory. 9, 10, 15

RSA Rivest–Shamir–Adleman. 13, 37, 38, 42

SDK Software Development Kit. 11

SECS SGX Enclave Control Structure. 9, 11–13

SGX Software Guard Extensions. 1, 2, 8–13, 15, 20–24, 26, 34, 36, 38, 45, 53–56, 59

SHM Shared Memory. 36

TCB Trusted Computing Base. 1, 2, 20–23, 26, 38, 53, 54

TCP Transmission Control Protocol. 52

TEE Trusted Execution Environment. 1

TLB Translation Lookaside Buffer. 13

UEFI Unified Extensible Firmware Interface. 54

VFS Virtual File System. 16, 17, 53

xattr Extended Attribute. 17, 18, 32, 35–37, 39, 53