Cross-platform Data Integrity and Confidentiality with Graduated Access Control

by

Feifan Chen

BASc Engineering Science Electrical and Computer Engineering, University of Toronto, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

December 2016

Abstract

Security of data is tightly coupled to its access policy. However, in practice, a data owner has control of his data's access policies only as far as the boundaries of his own systems. We introduce **graduated access control**, which provides mobile, programmable, and dynamically-resolving policies for access control that extends a data owner's policies across system boundaries. We realize this through a novel data-centric abstraction called **trusted capsules** and its associated system, the **trusted data monitor**.

A trusted capsule couples data and policy into a single mobile unit. A capsule is backwards-compatible and is indistinguishable from any regular file to applications. In coordination with the trusted data monitor, a capsule provides data integrity and confidentiality on remote devices, strong authentication to a trusted capsule service, and supports nuanced and dynamic access control decisions on remote systems.

We implemented our data monitor using ARM TrustZone. We show that graduated access control can express novel and useful real world policies, such as revocation, remote monitoring, and risk-adaptable disclosure. We illustrate trusted capsules for different file formats, including JPEG, FODT, MP4 and PDF. We also show compatibility with unmodified applications such as LibreOffice Writer, Evince, GpicView and VLC. In general, we found that applications operating on trusted capsules have varying performance, which depends on file size, application access patterns, and policy complexity.

Preface

The work presented in this thesis was conducted by the author in the Network, Security and Systems Lab under supervision of Dr. Ivan Beschastnikh.

Table of Contents

List of Tables

List of Figures

Glossary

TEE Trusted Execution Environment

OS Operating System

Acknowledgments

First, I would like to thank my supervisor, Dr. Ivan Beschastnikh, for all his help and guidance throughout the entirety of the project, without which this would not have been possible. I am also grateful to Dr. Andrew Warfield who provided valuable guidance in pushing the project forward and Dr. Norman Hutchinson for being second reader for my thesis.

Secondly, I would also like to thank my labmates in the NSS lab (Networks, Systems, and Security) for keeping me company. Especially I would like to thank Amanda Carbonari who helped me finish my thesis on time. I would also like to thank Haoran Yu and Thomas Liu for their contributions to the project.

This work was supported by the NSA National Information Assurance Lab and by the UBC Computer Science Department.

Chapter 1

Introduction

Today's documents, pictures, videos, and other user-generated data are highly mobile and only rarely live on a single device. This data is shared, backed-up, replicated, synced, and in general moved from one device to another. While a datum's owner has full control over the data access control policies on his set of devices, once his data has moved to a device he does not control, the data comes under the access control policies set by the owner of the remote device. Paradoxically, the data owner, who has the most vested interest and knowledge about the security requirements of his data, gets *no say over the access control policies for his data on remote devices.* This is represented visually in Figure 1.1(a): the owner's access control circle includes just his device(s).

As an example, consider how Bob might share a photo with Alice. Bob first

(a) **(b)**

Figure 1.1: (a) Device-centric access control, and **(b)** data-centric graduated access control.

uploads the photo to Dropbox and then grants Alice access to the photo. With this small action the access policy on his photo in the cloud changes to the policies set by Dropbox. Further, once Alice downloads the photo, Alice's copy of Bob's photo has yet another set of policies, dictated by Alice's device and software stack. In this scenario, Bob must blindly trust Alice to both know and adhere to Bob's desired access policies for his photos. For example, this may include maintaining the photo's privacy by not opening it at public places or deleting it when Bob asks her to at a later point in time. Further, Bob also becomes vulnerable to any compromise of Dropbox.

So, how can Bob ensure that his photo's access control policies are met when he shares the photo with Alice and uploads it to Dropbox, both of whom Bob do not trust?

Existing commercial approaches offer one answer: extend Bob's system boundary across remote devices, including Dropbox servers and Alice's devices. These approaches require intrusive modifications to the remote software stack, either to the platform [5, 9] or to the application [8], and are variants of the VPN model: control data by not permitting it to leave the data owner's system boundary. Academic solutions offer other variants, such as DIFC [17, 19, 31, 36, 38, 43, 44] and taint-based solutions [21, 22, 26, 45], which have focused on ensuring data security across different levels of the local stack, but like the previously described commercial solutions, assume a single administrative domain.

Previous solutions are predominantly process-centric. That is, data access policy is controlled by the applications and the platform. This design is in conflict with the fact that it is the *data* that is mobile. In this model, control of data access policy is tied with data distribution — once the data leaves the data owner's systems, he can no longer control how his data is accessed.

In this work, we present a data-centric solution called **graduated access control** and realize it using a trusted capsule abstraction that ties policy to data, and a secure data monitor for secure policy evaluation on remote devices. With graduated access control a user can define data access policies on a continuum rather than a binary can/cannot. Access is resolved dynamically and in a fine-grained manner (in our implementation policies are evaluated at system call granularity). Policies are stateful programs that can base access decisions on information like

location, time, and number of prior accesses. Figure 1.1(b) illustrates graduated access control as a data-centric extension of a data owner's device-centric access control.

Our realization of graduated access control uses an abstraction called trusted capsules, which consists of the data and a policy encapsulated into a single mobile unit. This pairing ensures that control over the data in a trusted capsule is no longer dependent on where and how the capsule is distributed. Capsules also provide backward compatibility with existing applications (a capsule is a regular file) and support a diverse set of policies that can be defined on remote devices as first-order abstractions. Trusted capsules allow the data owner to express policies related to risk-adaptable disclosure, compliance, revocation, data provenance, and data integrity.

To enforce policies dictated by trusted capsules, we designed a data monitor. The monitor operates at data-level granularity in a trusted execution domain within ARM TrustZone [11], and transparently mediates access to data in the trusted capsule by applications on remote platforms. Our data monitor uses a pragmatic threat model that acknowledges the realities of today's mobile devices — data shown on the screen can be captured by cameras or read by third parties. The data monitor, therefore, provides secure policy evaluation for data declassification on the remote device. But, once the data is declassified the monitor makes no attempt to track or control data movement, e.g., it does not prevent an application that was granted data access from leaking the data.

We show that graduated access control and trusted capsules can express real world policies for different file formats, including JPEG, FODT, and PDF. We also show that unmodified applications, such as LibreOffice Writer, PDF reader Evince, and video player VLC, work with capsules and can take advantage of graduated access control. When using a capsule containing a JPEG, our system imposes a space overhead of 1.42% and a slowdown of 45x when opening the image. This overhead depends on how a policy is expressed and the data is being accessed. We also measured the read/write throughput and latency. A null policy has a throughput of 1.4 MB/sec, compared to the 500 MB/sec throughput without our system. The latency per syscall operation varies but imposes an average slowdown of 391x, although this does not reflect the perceived application slowdown.

Our contributions can be summarized as follows:

- A data-centric graduated access control abstraction for protecting sensitive data across system boundaries.
- A backwards compatible trusted capsule abstraction and data monitor, which transparently support flexible policies at data-level granularity on remote devices.
- An evaluation of the performance and expressiveness of our system.

In the following, we build upon the motivation set forth in this chapter by providing real world use cases in Chapter 2. Then we provide an overview of our Trusted Execution Environment (TEE) in Chapter 3, the design and implementation of our trusted capsules and its associated data monitor in Chapter 4 and 5, and the summary of our results in Chapter 6. We discuss related work in Chapter 7 and future work in Chapter 8.

Chapter 2

Use Cases

In this section we introduce four use-cases which we use to motivate the need for graduated access control, and to structure our evaluation (Chapter 6).

Sensitive documents. Employees such as lawyers, government administrators and corporate executives routinely bring home sensitive documents (e.g., merger documents) to review on their own devices. Usually, this convenience is in tension with the employer's need to protect sensitive information. The employer may wish to attach risk-adaptable policies to sensitive documents, such as policies to redact certain sensitive document regions based on location, time, and the employee's role within the organization. For example, a law firm may want to constrain access to the merger document to within the confines of the office and to a lawyer's home. Outside of these geographic boundaries, the law firm may wish to redact the principals of the merger from the text of the document when it is opened as a graduated risk-adaptable policy. This prevents a careless lawyer from reviewing the sensitive document in a public space where there may be curious eyes.

Identity protection. Social Security Numbers (SSN) in the USA have evolved into a national personal ID, making it an identity theft target. Many US universities display student SSNs on college transcripts [10]. These transcripts must be shared with other organizations, for example, to prove registration status. These transcripts may then be kept on file for undetermined lengths of time. A malicious employee or a compromised system may then reveal a student's SSN information at a later point in time. A student may want to attach risk-adaptable policies that

stops access after a reasonable period of time has elapsed.

Sharing photos & videos. Celebrities, such as the British royal family, need to protect their private photos and videos [7]. However, personal data in the cloud is vulnerable to human engineering efforts and attacks on the cloud's software stack, such as when the photos of the royal family were stolen from the iCloud [6] of one royal family member. Ideally, if such personal data were to fall into the wrong hands, a data policy would prevent access. Further, in the event of a stolen device or changed social circumstance (e.g., break-up), the data owner can revoke access by retroactively changing the access policy.

E-Health records (EHR). It has long been the ambition of any health care authority to establish a national record of each citizen's health history to improve patient care. However, the centralization of sensitive health information has numerous security concerns [12]. These include individuals who abuse privileged medical information for personal gain and medical professionals who retroactively change medical diagnosis to cover up mistakes. Trusted capsules allow a distributed model for EHR that would mitigate these concerns. The health authority controls the EHR's policy while the patient physically carries the EHR – similar to a physical healthcare card. Devices belonging to health-care professionals and patients are provisioned with role-based identities by the HA. In this de-centralized model, only health care professionals with the health-care provider role who are given *physical* access to the the patient's EHR would have access.

The HA may specify policies that only allow the EHR to be append-only to prevent retroactive modifications. It can attach policies that only allow the current doctor to view the health record. Further, they may have redact policies that hides a top-level watermark for any person with the role of patient but is not redacted for any person with the role of doctor to ensure the provenance and the integrity of the distributed EHR.

Chapter 3

TrustZone & OP-TEE Overview

Trusted capsules allow advisory policies to be enforced on remote devices that the data owner does not control. To protect sensitive operations such as trusted capsule policy evaluation from remote users who can run an arbitrarily software stack, we require a TEE that is resistant to potential compromise of both applications and OS running on the remote device. We use ARM TrustZone technology as our TEE and Linaro OP-TEE as our TEE low-level software stack. Within this TEE, we handle sensitive cryptographic operations, perform policy evaluation, securely store policy state, and anchor a secure channel to the policy coordinator.

3.1 TrustZone

ARM TrustZone [11] is widely available on current commodity ARM processors. TrustZone physically partitions the CPU, memory and peripherals into two isolated logical "worlds" – normal and secure. Each world has its own banked system registers and MMU. To isolate the two worlds, all communications must pass through a small and heavily verified *secure monitor* gate. To facilitate a *world switch*, a special *smc* instruction is used to trap into the secure monitor. The secure monitor saves the banked registers (e.g., return address, stack pointer) of the calling world and loads the banked registers of the callee world prior to executing *eret* to return to the last execution point in the callee world.

Where the *smc* traps to is controlled by the secure world through its exception

	Secure	Normal
EL0	Trusted Application	Application
EL1	Secure Operating System (OS)	Normal OS
EL3	Secure Monitor	-

Table 3.1: ARM processor modes.

table register – VBAR, which holds the memory address of the exception table. The memory that holds the exception table can also only be accessed by the secure world.

The ARM TrustZone security model provides the following hardware-based guarantee: **the normal world cannot access the registers, memory or peripherals assigned to the secure world; but the secure world can access normal world registers and memory**.

For registers, this guarantee is provided through a Secure-Modify-Only NS-bit in the ARM System Control Register (SCR), which controls the world-view for banked registers. Control of this bit is retained exclusively by the secure world enabling it to access banked system registers of both worlds, but not vice versa for the normal world.

For memory, the secure world provides such a guarantee by either taking exclusive control of on-chip memory such as secure SRAM [4] or by mapping a section of the general off-chip memory and hiding it from the MMU of the normal world.

For peripherals, secure and normal world access are partitioned by interrupt modes. ARM processors contain two interrupt modes – FIQ and IRQ. Each interrupt mode can be individually assigned to trap to code in the normal or secure world. Therefore, a peripheral can be assigned to a specific world by assigning it to the corresponding interrupt mode. The usual set-up assigns FIQ to the secure world and IRQ to the normal world, as most existing normal world drivers currently operate using the IRQ mode.

For additional hardware protection for off-chip memory and device protection, additional hardware, such as TrustZone Protection Controller (TZPC) and TrustZone Address Space Controller (TZASC), can be added to extend the dual-world abstraction to the AXI-bus, memory controllers and interrupt controllers.

The secure/normal paradigm operates orthogonally to the traditional concept of privilege levels, see Table 3.1. The secure monitor operates in secure mode at the highest privilege level (EL3), while untrusted application code and privileged normal world OS operate in non-secure mode. The secure mode at privilege level EL0 and EL1 is reserved for trusted applications and the trusted OS.

Architecturally, the privilege level of the CPU is controlled by a system register called Saved Program State Register (SPSR). The SPSR register is banked between different modes of operation for the ARM processor and is saved/reloaded during a world switch before returning to the point of last execution. The current SPSR in use is loaded into the Current Program State Register (CPSR).

TrustZone enables the applications and the OS running in the secure World to remain protected even if the normal world OS or applications are arbitrarily compromised.

3.2 Linaro OP-TEE

Linaro OP-TEE is an open-source software stack for ARM TrustZone. It provides a secure world OS (OP-TEE OS) for executing trusted applications, a low-level secure monitor for world-switching (ARM Trusted Firmware) and a TrustZone driver (OP-TEE Linux driver & OP-TEE Supplicant) for the normal world OS, such as Linux, to access TrustZone and execute secure world RPCs. We use Linaro OP-TEE as-is except for our custom extensions that enable direct access to the network and the file system as RPCs by trusted applications in the secure world. These secure world RPCs are executed by OP-TEE Supplicant which runs in normal world user space as a single threaded application and are intermediated by OP-TEE Linux Driver in the normal world kernel.

The following description is based on the HiKey system-on-chip (SoC) with Linux as the normal world OS.

3.2.1 ARM Trusted Firmware

ARM Trusted Firmware (ATF) [2] is a set of reference boot and runtime firmware designs for ARM TrustZone. It initializes the secure world through a multi-staged boot sequence, as shown in Figure 3.1. A root-of-trust can be built by

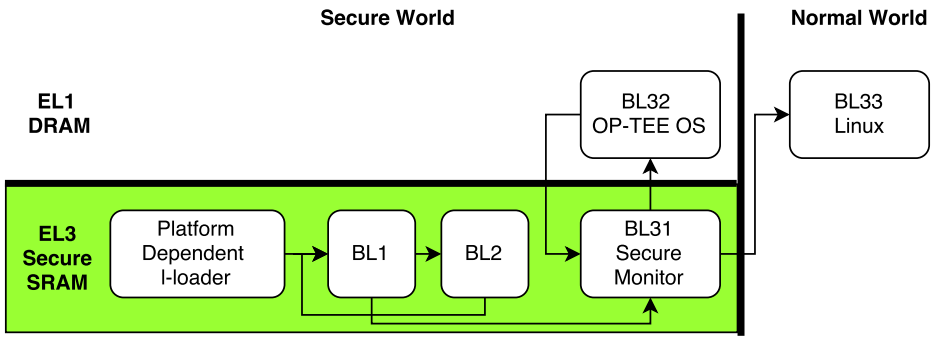


Figure 3.1: ARM TrustZone Boot Sequence.

having each stage attest the image of the next.

3.2.2 OP-TEE OS

OP-TEE OS is capable of multi-threading, memory management, and running and isolating trusted applications. OP-TEE OS does not have a scheduler. It operates as a slave in a master-slave relationship with the normal world OS. Therefore, OP-TEE OS can only simultaneously run as many trusted application instances as there are cores at any given time. On multi-core architectures, each CPU can independently perform a world switch. When an interrupt occurs that needs to be handled by the normal world, OP-TEE OS transitions back into the normal world and once the interrupt has being handled, returns to its last point of execution within the secure world. Communication between the normal world and secure world occurs through a piece of pre-allocated shared memory accessible by both worlds. The shared memory is allocated by the secure world but is managed by the normal world. The secure world OS may access peripherals under the normal world's control and allocate shared memory through RPC calls into the normal world. For example, OP-TEE OS uses these RPC calls to access the normal world file system, with which it implements secure storage using a provisioned root key.

OP-TEE OS provides useful abstractions to build trusted applications that run in secure world user space (Secure EL0). Trusted applications can be single-instance or multi-session. OP-TEE OS applications conform to the GlobalPlatform Internal API [3] where each trusted application must implement a set of well-

Internal API	Client API	Function
CreateEntryPoint	InitializeContext	Initialize a context in TrustZone driver
DestroyEntryPoint	FinalizeContext	Deletes a TrustZone context
OpenSessionEntryPoint	OpenSession	Creates an instance of the trusted application
CloseSessionEntryPoint	CloseSession	Destroys an instance of the trusted application
InvokeCommandEntryPoint	InvokeCommand	Call one of trusted application's functions
-	RegisterSharedMemory	Registers a chunk of memory for use between the two worlds
-	AllocateSharedMemory	Allocate a chunk of memory from the shared memory pool
-	ReleaseSharedMemory	Free a chunk of memory allocated from the shared memory pool
-	RequestCancellation	Request an instance of trusted application to stop and return

Table 3.2: Linaro OP-TEE API. Internal APIs are used by trusted applications and are prefixed by *TA_*. Client APIs are used by the normal world and are prefixed by *TEEC_*.

defined functions as entry-points. Client applications in the normal world invoke these functions through a similar set of GlobalPlatform Client API [3]. The list of functions are listed in Table 3.2. We use these APIs and secure storage provided by OP-TEE OS to build our multi-session trusted capsule application at the core of our trusted capsule monitor. Any call into trusted applications from the normal world are serialized on the normal world side by the TrustZone device driver.

3.2.3 OP-TEE Linux Driver

OP-TEE Linux Driver provides the normal world OS (Linux) access to TrustZone. It represents TrustZone as a device file, which can be accessed from the normal world through the set of APIs listed in Table 3.2 from both user and kernel space. The TrustZone driver is responsible for two main tasks – (1) calling into trusted applications running in TrustZone and (2) handling RPC requests from

OP-TEE OS (e.g., file system, shared memory allocations). For trusted capsules, we extended the limited set of RPC calls available to the OP-TEE OS to include networking and direct file system operations. The TrustZone driver executes RPCs by using the OP-TEE Supplicant.

When the TrustZone driver calls into the secure world, it uses two unique identifiers – ”session object” and ”function ID”. Each trusted application instance is represented by a ”session” and each function that the trusted application can perform by a ”function ID”. Together, these two identifiers specify the entry point for the call into secure world. Function parameters are passed by value or by reference through shared memory between the two worlds.

3.2.4 OP-TEE Supplicant

OP-TEE Supplicant takes RPC invocations from OP-TEE Linux Driver and executes the equivalent system calls through the normal world OS to access the relevant peripheral devices. These peripheral devices can include file system block devices and network cards for I/O. Linux *dmabuf* and *mmap* are used to pass data between the user space OP-TEE supplicant and kernel space OP-TEE Linux Driver. Only a single instance of the OP-TEE supplicant can run at any given time and this is enforced by the OP-TEE Linux Driver. We do not intercept any systems calls made by the OP-TEE Supplicant running in normal world user space. The OP-TEE Supplicant never accesses decrypted trusted capsule data and it cannot write to a capsule without the corruption being detected.

Chapter 4

Design

A trusted capsule consists of some data and the policy for accessing the data, both encapsulated into a single encrypted file. The layout of a trusted capsule is shown in Figure 4.1. It consists of a header and multiple chunks of encrypted content. The header identifies a file as a trusted capsule and contains the size of the trusted capsule and a unique ID. The content of the trusted capsule is composed of *chunks* that consist of both the data and policy encrypted together. Each chunk has its own hash. A hash of all the hashes is also stored in the header. We break the content of the trusted capsule into chunks so that a modification of any one part of the trusted capsule's data does not require the entire file to be re-hashed and re-encrypted. We assume that role-based credentials and cryptographic keys associated with the trusted capsules are loaded into TrustZone's secure storage through a tertiary channel.

Figure 4.2 illustrates our data monitor system design. We illustrate the flow of data and control between the various data monitor components for a *read* system call in Figure 4.3.

The trusted capsule data monitor consists of multiple components located across both secure and normal worlds. In the normal world, we implemented a system call interceptor that redirects operations on trusted capsules to the trusted capsule application. By mediating access to trusted capsules within the secure world, we are able to securely and dynamically evaluate the programmable policies that they carry across system boundaries. The system call interceptor operates in a transpar-

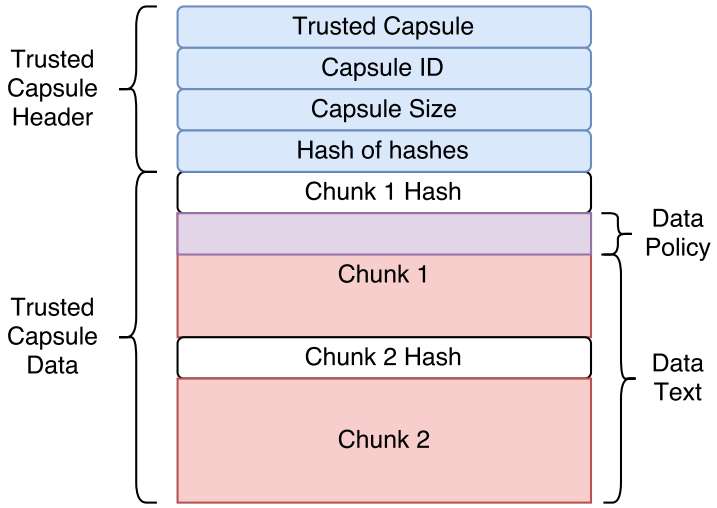


Figure 4.1: Trusted capsule layout.

ent manner, enabling compatibility with existing applications.

In our data monitor implementation, we do not trust the normal software stack. Therefore, we make no guarantees on data that is decrypted to the normal world. We consider any decrypted data revealed to the normal world as declassified. Further, while dynamically resolving trusted capsule policy, we consider any local state obtained through the normal world (e.g., GPS, time, system state) as also untrusted. Therefore any policy that adjudicates access based on such information is an advisory policy and cannot be strongly enforced.

To make stronger guarantees would require intrusive modifications of the remote software stack and limit our solution to specific classes of applications or data types – impacting backward compatibility and generality. Therefore, our trusted data monitor design point is based on a conscious decision to take a more pragmatic approach. We provide only limited and advisory declassification capabilities, such as limiting network and file system access, post-declassification of trusted capsule data. Further, we acknowledge to data owners that such policies that use the local device state are advisory in nature. Policies based on remote state from trusted capsule server or local state in the trusted capsule's secure storage, however, are secure sources of information for policy evaluation.

In the following, we describe key components of our trusted capsule data mon-

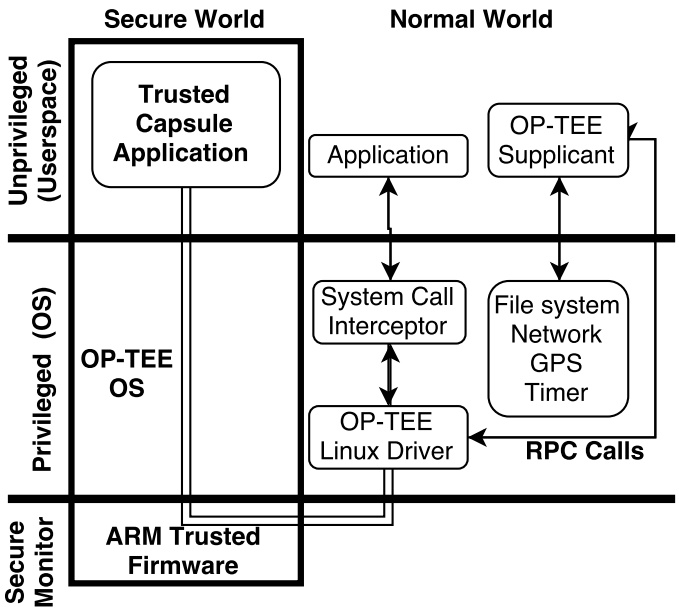


Figure 4.2: Trusted capsule data monitor system model. The secure world trusted capsule applications access peripheral I/O through RPC calls to the OP-TEE Supplicant via the OP-TEE Linux Driver. Application system calls that affect trusted capsules are intercepted and forwarded to the trusted capsule application through the system call interceptor and OP-TEE Linux Driver.

itor and how it creates an ecosystem for enabling uniform, programmable, and dynamic policies along with data mobility.

4.1 Trusted Capsule Application

At the core of the trusted capsule monitor is the trusted capsule application running in the secure world. At a high level, a trusted capsule application implements function methods that perform specific tasks. These function methods can be called directly from the normal world once the trusted capsule application instance is created. Only a single function may be executed at a time from the normal world. These function methods execute synchronously and run to completion, although during the execution, the trusted capsule application may temporarily switch to the

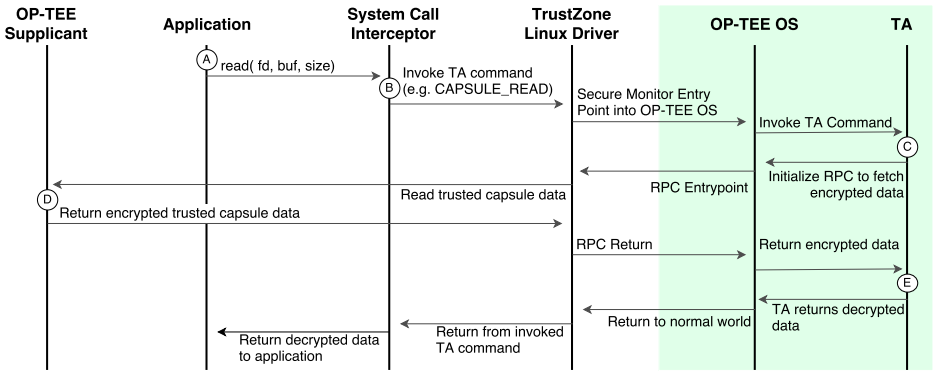


Figure 4.3: Trusted capsule monitor operation (shaded region operates in the secure world). **A.** Application *read* system call is intercepted. **B.** Interceptor calls into secure world to invoke CAPSULE_READ. **C.** The trusted capsule application (TA) evaluates the *read* policy. It initializes RPC calls to fetch encrypted trusted capsule data. **D.** The RPC call is executed by OP-TEE Supplicant and the results are returned to the secure world. **E.** TA decrypts the encrypted data and applies any redactions. It then returns the decrypted data to the normal world application.

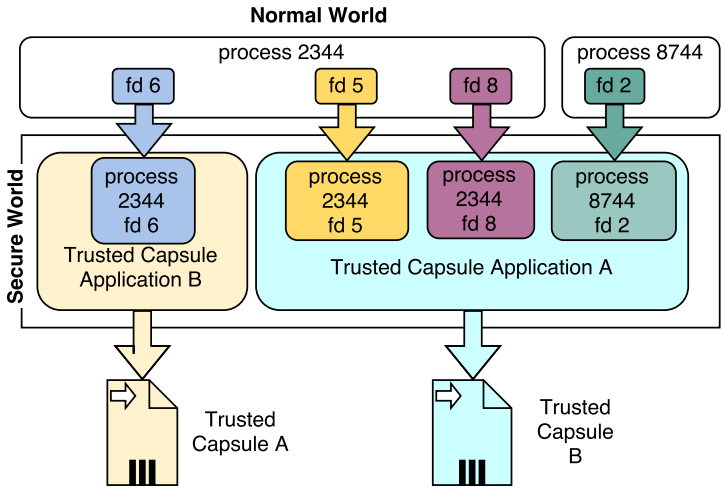


Figure 4.4: Trusted capsule monitor session model. Each Trusted Capsule Application maintains the session for a single trusted capsule. It supports multiple file handles per trusted capsule. The file handles consists of the process ID and file descriptor.

normal world context to perform RPC or handle normal world interrupts.

The trusted capsule application maintains a runtime session for each trusted capsule. Each trusted capsule application internally maintains its own cryptographic and hashing handles, role-based credentials, hashes of the trusted capsule contents and the file offsets of the policy and data sections. Functionally, it evaluates the policy of the trusted capsule, performs cryptographic and hashing computations, and stores secure persistent state associated with the trusted capsule. Further, the trusted application acts as the endpoint for secure communication with a trusted capsule server for performing remote actions (e.g., fetching remote state or initiating a policy change). The trusted capsule application handles sensitive information and performs critical functions that must be protected against the untrusted normal world. Therefore we require this part of the trusted data monitor to execute within TrustZone.

Trusted capsule can be accessed simultaneously by multiple processes. We maintain a map of the process ID and file descriptor tuples to their current data offset in the trusted capsule within each trusted capsule application instance, as shown in Figure 4.4.

We use OP-TEE OS native secure storage capability to store our cryptographic keys and persistent trusted capsule states. Cryptographic information is stored in serialized binary while trusted capsule states are stored in key-value format. All trusted capsule encryption keys are stored in a single secure key file. We allow the key file to be accessible by multiple trusted capsule applications at a time so that multiple sessions can be instantiated simultaneously to handle different trusted capsules. In contrast, each trusted capsule get its own secure persistent state file. Persistent state files can only be opened by the trusted capsule it is associated with and by only a single trusted application at a time. This is enforced through the OP-TEE OS and enables us to strongly enforce only a single trusted capsule instance to handle each trusted capsule and each trusted capsule to only be able to access its own persistent state file.

Trusted application methods are invoked by the system call interceptor based on specific system calls. In this way, access control can be enforced dynamically based on the local and remote state when such a system call is initiated. We describe several of these methods below.

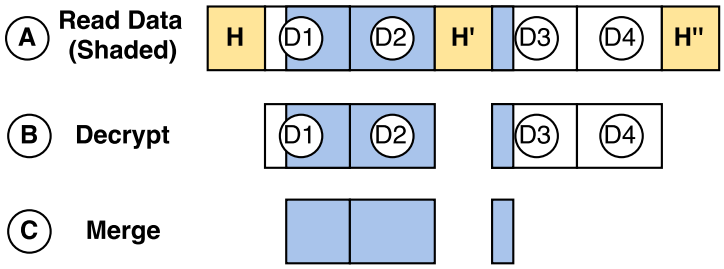


Figure 4.5: Trusted capsule read of a capsule (shaded region). A read may overlap several chunks (e.g., D1 and D2) and may not be aligned to the cryptographic key size (e.g., D1 and D3).

CAPSULE_OPEN. This function is invoked by the *open* system call. The function performs several key tasks – (1) it finds the cryptographic keys for the trusted capsule, (2) parses the trusted capsule policy and data, (3) verifies the capsule contents and (4) loads and evaluates policy.

CAPSULE_READ. This function is invoked by the *read* system call and its variants. A read operation contains several stages – (1) we evaluate the *read* policy, (2) we fetch the encrypted trusted capsule data, (3) recalculate the hash to check the integrity of each chunk involved in the read, (4) decrypt, merge and redact the data. A *read* may not be aligned on a chunk or key size boundary. In these cases, the actual *read* buffer must be merged. This process may involve multiple read RPC calls into the normal world for a single read system call. This process is shown in Figure 4.5.

CAPSULE_WRITE. This function is invoked by the *write* system call and its variants. Similar to *read*, a *write* may not be align with the chunk or key size. A singular *write* system call may also be broken into multiple independent aligned writes. To ensure the integrity of a chunk during the course of a write, we make modifications to a chunk in-memory. A write operation contains the following stages – (1) we evaluate the *write* policy, (2) we read in the entire chunk into secure world memory and verify its integrity, (3) we make the modifications in-memory, (4) recalculate the hash of the chunk and of the file, (5) re-encrypt the modified chunk and write-back the data and hashes to the underlying file.

CAPSULE_WRITE_EVALUATE. This function is called when a process makes

a *write* system call on another file or peripheral after having accessed a trusted capsule. It evaluates the *declassify* policy of the trusted capsule and takes as its input the stringified destination (e.g., "/home/user/text.txt" or "128.0.0.2:10") and outputs whether the policy allows the declassifying *write* to occur.

CAPSULE_LSEEK. This function is invoked by the *lseek* system call. It internally updates the data offset of the file handle identified by the process IDs and file descriptor.

CAPSULE_CLOSE. This function is invoked by the *close* system call. It internally removes the file handle identified by the proces ID and file descriptor.

4.2 Trusted Capsule Server

To provide data mobility yet maintain a uniform data owner policy, we use a trusted capsule server, owned by the data owner, as a central policy coordinator for one or more of the data owner's trusted capsules. Its role within the trusted capsule infrastructure serves two important functions.

First, it is a source of trusted information for policy evaluation that is secure against compromise of remote device where the trusted capsule resides. For example, a trusted capsule's policy may specify that the policy obtain its current time from the trusted capsule server instead of from the local device clock. Further, trusted capsules may contain policies that require state that does not exist locally.

Second, the trusted capsule acts as a point of administration for trusted capsule policy. The data owner can initiate policy changes, delete trusted capsules remotely, or receive information about trusted capsule activity on remote devices (e.g., whether a capsule has been opened).

The trusted capsule server's IP and port are specified by the trusted capsule policy. Using a trusted capsule server controlled by the data owner expands trusted capsule's policy capabilities at the cost of extra policy evaluation delay due to network latency.

The communication between the trusted capsule server and the trusted capsule application is mediated by the local device's normal world. To prevent man-in-the-middle attacks, all communication between these two endpoints are encrypted with a randomly generated nonce to protect against replay attacks.

Custom lua function	Description
getlocalstate(key)	Get the value of state associated with this key from secure storage
getdataoffset()	Get the data offset and length of the current operation (e.g. read/write)
getdatasize()	Gets the size of the trusted capsule data
getgps()	Get the longitude and latitude from the GPS device
gettime()	Get the current time as an integer since January 1, 1970
getserverstate(key)	Get the value of state associated with this key form the trusted capsule server
reportlocid()	Send the current location, time, identity, operation and data offsets to the trusted capsule server
checkpolicychange(version)	Check with the trusted capsule server for policy updates
setstate(key, value)	Set the value of state associated with this key in secure storage
delete()	Delete the trusted capsule file

Table 4.1: Lua policy extensions.

4.3 Lua Policy Engine

To support general programmable and dynamically-resolving policies, we embed a Lua interpreter into our trusted capsule application. Lua is an extensible, Turing-complete, and interpreted language that is designed for easy integration with C applications. It supports comparison and logical operators, arrays, conditional statements, integer and floating point mathematics, and loops. New functionality can be added to the Lua language in the form of custom functions.

We implement several key abstractions as custom functions (Table 4.1). These abstractions form the core of our Lua-based policy language. Figure 4.6 illustrates an example policy template.

We discusses the key features of our Lua-based policy language below.

Global Variables. Every Lua policy defines several key global variables. These variables include the IP address and port of the trusted capsule server, the replacement character to use for redactions, an array containing the byte ranges within the file to redact and the current policy version. These global variables are shown at the top of Figure 4.6. These variables are used as part of policy evaluation. For example, *checkpolicychange(version)* uses the current policy version set

in the Lua global *version* to check for updates.

Revocation. A Lua policy can specify revocation in two different ways. First, we allow the policy to instruct the trusted capsule application to delete the trusted capsule file. This can be invoked remotely or locally. When the *delete()* Lua function is called, we immediately overwrite the trusted capsule file with zeros. This is because the Linux OS does not actually delete the file until the file's reference count becomes zero. We then make an RPC call into the normal world to delete the file and destroy the trusted capsule application session. Such a revocation is permanent. Second, we allow retroactive policy changes via the trusted capsule server. In this scenario, the policy specifies a condition under which *checkpolicychange(version)* is called. If a new policy exists at the trusted capsule server, it is downloaded by the trusted capsule application. Policy changes are temporary as the owner could always change the policy back. With trusted capsules, revocation is not simply a binary, but *graduated* based on user needs.

Logging Trusted Capsule Activity. We extended the Lua language with the ability to report information to the trusted capsule server. The policy can specify arbitrary conditions under which the *reportlocid()* function is called. In this case, the trusted capsule application reports the location, operation, local time and identity to the trusted capsule server. This can be used to implement policies that inform the data owner about accesses to his data. The data owner may use this information as an audit trail or as a review mechanism to inform his policy decisions, such as revocation.

State. Policy evaluation uses the state of the local and remote systems. These states, at the time of policy evaluation, enable access control to be dynamically resolved. First, we provide the ability to read and write to the state file uniquely associated with the trusted capsule. This enables the Lua policy to use persistent state, such as role-based credentials. Further, it provides an extremely useful abstraction which can be used to express novel remote policies. For example, policies may specify a limited number of accesses. For security reasons, we do not provide access to the file that stores the cryptographic keys or state files of other trusted capsules. Second, we provide the ability to access state in peripheral devices. In our current implementation, this includes the local device clock and GPS device. This enables policies to be based on time and location. When combined with the

```
1 trusted_server = "198.162.52.244"
2 port = 3490
3 replace_char = "#"
4 redact = {};
5 version = 1;

7 function policy( op )
8  res = false;
9  pol_changed = false;

11  if op == 0 then
12   -- OPEN POLICY
13  elseif op == 1 then
14   -- READ POLICY
15  elseif op == 2 then
16   -- WRITE POLICY
17  elseif op == 3 then
18   -- DECLASSIFY POLICY
19  elseif op == 4 then
20   -- CLOSE POLICY
21  end

23  return res, pol_changed;
24 end
```

Figure 4.6: Lua policy template.

ability to persist state, this can create more complicated policies. For example, a policy may specify a check on a certain condition periodically (e.g., policy updates). Third, we allow the policy to access the session state within the trusted capsule application. This includes the current data offset, current length of the read or write function call and the stringified destination of a declassifying write. Finally, we also provide the ability to get information from a trusted capsule server. This provides an unique and trusted information source for policy evaluation.

4.4 System Call Interceptor

The system call interceptor enables applications to transparently access trusted capsules. The interceptor is dependent on the OP-TEE Linux Driver and interfaces

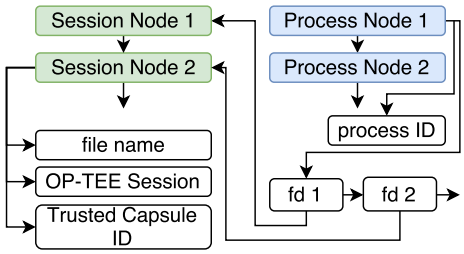


Figure 4.7: Session and process states used to keep track of trusted capsules by the system call interceptor.

with it through the same client-side kernel API as shown previously in Table 3.2.

Internally within the interceptor, we maintain a session hash table that represents a trusted application instance and a process hash table to keep track of the interceptor states. The session hash table contains session nodes that store the OP-TEE trusted capsule application instance handle associated with each unique trusted capsule based on their ID. The process hash table maintains a list of process nodes that store the trusted capsules accessed by a particular process. The relationship between the nodes of the these two hash tables is shown in Figure 4.7. The process nodes maintain a list of file descriptors that correspond to trusted capsules that they have accessed over their lifetime. These file descriptors store pointers to the sessions that represent each trusted capsule application within the session hash table.

To protect these data structures from concurrent access by multiple processes, we use mutexes and spinlocks to enforce exclusive access. The current locking mechanism is coarse, but can be improved with a more refined implementation. We do not enforce exclusive access to the OP-TEE Linux Driver as it already serializes all calls into TrustZone for a particular trusted capsule application session.

We briefly describe the actions of the interceptor for each system call we currently intercept.

open. We use the trusted capsule header to determine whether the system call operates on a trusted capsule. If it is, we first check the session hash table to see if a session for the trusted capsule already exists. If no session was found, we create one by calling into the secure world using the TEEC_OpenSession() API to create another instance of the trusted capsule application. We then invoke the trusted

application's CAPSULE_OPEN function. If the *open* policy of the trusted capsule is satisfied, a regular file descriptor is returned to the application. Finally, we add the new process and session node to the hash table if they do not already exist and update the session's reference count.

close. We check the file descriptor against the capsule file descriptor list in the process node to determine whether the file descriptor represents a trusted capsule. If it does, the interceptor calls the CAPSULE_CLOSE function. It then sets the file descriptor in the process hash table to -1. However, it is important to note that once the trusted capsule is closed by an application, it is not removed from the list of trusted capsules that the application has accessed. In this case, the declassification policy of the trusted capsule would still be evaluated for future writes as the application may still maintain decrypted trusted capsule data within its memory.

lseek. We check the file descriptor against the capsule file descriptor list in the process node to determine whether it is an operation on a trusted capsule. If it is, the interceptor calls the CAPSULE_LSEEK function. The trusted capsule application internally updates the file position.

read. We check the file descriptor against the capsule file descriptor list in the process node to determine whether the read is on a trusted capsule. If it is, the interceptor calls the CAPSULE_READ function. If the *read* system call policy of the trusted capsule is satisfied, the trusted capsule application returns the decrypted data, although the data may be redacted.

write. System calls that declassify information, such as *write*, affect trusted capsules even when they are not directed at trusted capsules themselves. Therefore, we must iterate through the process node's capsule file descriptor list to evaluate the declassification policy for each trusted capsule the process had accessed. We accomplish this by calling the secure world CAPSULE_WRITE_EVALUATE function of each trusted capsule. If the *write* is allowed by all the trusted capsules' *declassify* policy, the *write* is performed. In the case that the *write* is on a trusted capsule, it calls the CAPSULE_WRITE function. Otherwise, it falls through as a normal *write* system call.

exit. We check the process node's file descriptor list to determine all the trusted capsules that this process had accessed. For each trusted capsule, we call the TEEC_CloseSession API to remove the trusted capsule application in the secure

world if its reference count has decremented to zero.

4.5 Security

We consider two important security aspects of our trusted capsule monitor.

Trusted Capsules. Operations on trusted capsules are mediated by the trusted capsule monitor system. Any operations on the trusted capsules are forwarded to the secure world trusted capsule application for execution. Therefore, the confidentiality and integrity of the data within the trusted capsule is protected (at the hardware-level) even against compromises of the normal world OS. A compromised normal world OS may corrupt a trusted capsule, but the corruption will be detected.

Further, persistent trusted capsule state is stored in secure storage by OP-TEE. A compromised system may delete the encrypted files that contains the cryptographic keys or trusted capsule states, but this will also render the system incapable of accessing the trusted capsule.

Isolation between trusted capsule applications is enforced by the secure world OP-TEE OS. We further isolate each trusted capsule by having unique trusted capsules instantiate its own instance of the trusted capsule application and provisioning it with its own uniquely accessible file for its own persistent states.

Policy Evaluation. We consider attacks on the normal world from the executable policy running within the trusted capsule application. Our Lua-based policy language can contain arbitrary execution such as loops. It, therefore, may perform denial-of-service attacks by never returning from policy evaluation or exfiltrate information to its trusted capsule server. We are mindful of this fact and use the Lua interpreter as a sandbox. We disabled any Lua library that would allow the interpreter to interact with external systems (e.g., I/O, packages, debug and OS). In its place, we extended the Lua interpreter with functions that have much narrower scopes. Our extensions to the Lua interpreter cannot interact with any files other than the trusted capsule. Peripherals are read-only by Lua-based policies. Further, Lua-based policies can only read and write to its own state file. It cannot for example, read the key file – preventing the exfiltration of keys of other trusted capsules. Further, denial-of-service attacks can be stopped as the normal world can cancel a

command that does not return promptly through the TEEC_RequestCancellation() API, which stops long-running trusted capsule applications and returns immediately to the normal world.

Chapter 5

Implementation

We prototyped our system using the LeMaker HiKey [4] development board, which comes with 8 ARM Cortex-A53 processors, 8 GB of eMMC Flash, and 1 GB of RAM. The board has TrustZone unlocked.

We modified the Linaro OP-TEE OS version 1.0 to be our TrustZone software stack and deployed our system call interceptor as a loadable kernel module. In our current implementation, we intercept *open*, *read*, *write*, *lseek*, *fstat*, *close*, *exit* system calls and their variants. All intercepted system calls that do not affect a trusted capsule operate as-is. In order to intercept these system calls, we rootkit the normal world OS system call table and replaced the function pointers with our interceptor equivalents. Our modifications to the OP-TEE components across the software stack in both worlds consist entirely of additional RPCs that enable trusted capsule applications to access the file system and network directly. These modifications are minimal compared to the original OP-TEE code base.

As our HiKey board does not have a GPS, we built a virtual GPS device within OP-TEE Linux Driver that returns predefined longitude and latitude values. In the normal world, we run a pre-alpha release of a custom HiKey Debian OS based on Linux kernel 3.18.0. We used 128-bit AES and SHA-256 for encrypting and hashing trusted capsules.

Our current implementation supports the following composable actions based on policy evaluation:

- Allowing or denying the operation on the trusted capsule.

- Initializing or modifying persistent state (e.g., counter).
- Performing system-level byte-based redaction that partially discloses trusted capsule data. A data-type specific pre-processor can be used to generate the byte offsets.
- Reporting the operation, location, time, role-identity of the device to a global policy coordinator.
- Hard access revocation by deleting the capsule, triggered locally or remotely; and, soft revocation by arbitrarily changing the policy.
- Prevent declassification through the network or file system.

In total, our trusted capsule application has 3.2K lines of non-comment C code and our system call interceptor has 1K lines of non-comment C code.

Chapter 6

Evaluation

We evaluated four aspects of our system: **(1)** the utility and simplicity of the policy language, **(2)** the storage overhead of trusted capsules, **(3)** latency at the system call layer, and **(4)** perceived latency at the application layer.

All performance evaluations were performed on our HiKey development board. For the evaluation we used real file types and applications, including JPEG/GpicView image viewer, MP4/VLC video player, FODT/LibreOffice Writer, and PDF/Evince PDF reader.

6.1 Policy language

In our policy language evaluation we aimed to answer two questions: is the policy language adequate for expressing useful policies? Further, are these policies easy to express?

We answered our first question by writing trusted capsule policies for the example use-cases from Chapter 2. For our second question, we measured the LOC for each policy that we wrote and show the result in Table 6.1.

The ability to easily express complex policies tersely is important both as a proxy of simplicity and to bound the memory overhead of the Lua interpreter in the secure world. We found that with a few lines of code we were able to express complex policies such as redaction and revocation.

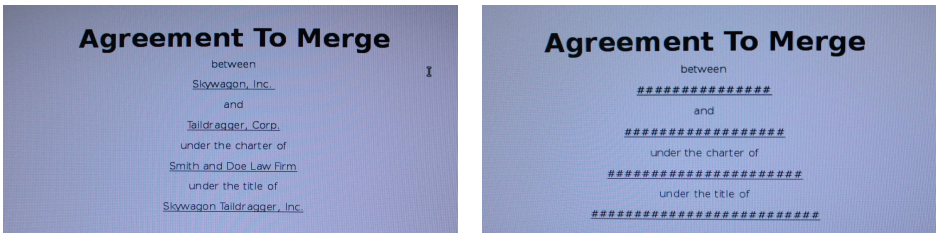
We focus on two such policies. The first policy is used in the context of a sensi-

```
1 ...
2 replace_char = "#"
3 redact_sensitive = {45379, 45393, 45532, 45549, 45705,
     45726, 45880, 45905, 46081, 46094, 46178, 46185,
     46293, 46309, 46380, 46385, 46449, 46458, 46528,
     46533, 46606, 46609, 46676, 46682, 46768, 46769,
     46835, 46844, 46953, 46963, 47124, 47141, 47225,
     47235, 47343, 47348, 47419, 47427, 47491, 47496,
     47571, 47586, 47659, 47662, 47729, 47735, 47821,
     47822, 47888, 47897, 48006, 48018, 48682, 48684,
     48751, 48757, 48843, 48847, 49003, 49008, 49705,
     49715, 49926, 49928, 50077, 50079, 50136, 50139,
     50950, 50957, 51078, 51080, 51266, 51268, 51325,
     51328, 52810, 52823, 54542, 54559};
4 redact = {};
5 ...
6 -- READ
7 elseif op == 1 then
8   local long, lat = getgps();
9   if((lat - 2130 >= 10) or (2130 - lat >= 10)) or ((lat
       - 22223 >= 10) or (22223-lat >= 10 )) then
10    redact = redact_sensitive;
11  end
12 end
13 ...
```

Figure 6.1: Sensitive merger document policy.

tive merger document. We highlight one component of the policy, the system-level redaction policy in Figure 6.1. This policy redacts all data within the byte offsets specified by *redact_sensitive*. These byte offsets were translated by a policy preprocessor from $<$ *Sensitive* $>$ XML-like tags used to define the merger document. This redaction only occurs if the document is opened outside of the office, as determined by the GPS coordinates. The original data is replaced with the character defined by *replace_char*. We show the results in Figure 6.2.

The second policy is for photos stored in the cloud. The policy, shown in Figure 6.3, allows only devices provisioned with the role-based identity "Kate" to access the photo. As described previously, with trusted capsules, an attacker who gains access to photos will not be able to access the capsule contents. On every



(a) Unredacted merger document. **(b)** Redacted merger document.

Figure 6.2: Before vs. after redaction.

```
1 ...
2 if getlocalstate( "cred" ) ~= "kate" then
3   res = false;
4 end
5 -- OPEN
6 if op == 0 then
7   view_status == getserverstate( "delete?" );
8   if ( view_status == "true" ) then
9     delete();
10  end
11 end
12 ...
```

Figure 6.3: Private image policy.

open system call, the monitor checks with the policy coordinator for updates to a remote state (line 7 in Figure 6.3). If the remote server returns *true*, the photo is deleted from the device.

For these and other policies we found that the Lua policy interpreter needed less than 2KB of allocated memory.

Policy	LOC
Merger Document	24
Transcript	25
Royal Image	30
EHR Patient	41

Table 6.1: LOC for example policies from Chapter 2.

	Data (KB)	Capsule (KB)	Overhead
PDF Doc	137.34KB	139.38KB	1.42%
JPEG Image	204.10KB	207.00KB	1.42%
MP4 Video	4142.40KB	4175.94KB	0.80%
LibreOffice Doc	54.80KB	56.70KB	3.47%

Table 6.2: Storage overhead for test data files.

6.2 Storage overhead

Converting regular data into trusted capsules incurs storage overhead that is proportional with the policy size and chunk size. We evaluate the associated overhead for different types of data.

Table 6.2 lists the storage overhead for a PDF, JPEG, MP4, and FODT files used in our evaluation. These data types were converted to trusted capsules with 4KB chunk sizes. We found the storage overhead to be negligible.

6.3 System call microbenchmarks

In considering system call level microbenchmarks, we focus on three questions.

Are operations on regular files affected? We measured the performance of system call operations with and without the system call interceptor for both latency and throughput under similar conditions. We found that the performance of system calls on regular data is not impacted, except for the *open* syscall. This is due to the overhead of checking whether the target file is a trusted capsule. The results are shown in Figure 6.4a and 6.4b.

In addition, once a process has accessed a trusted capsule, write throughput on regular data also decreases linearly with number of trusted capsules accessed by that process due to the need to evaluate each trusted capsule's declassify policies. The results are shown in Figure 6.5.

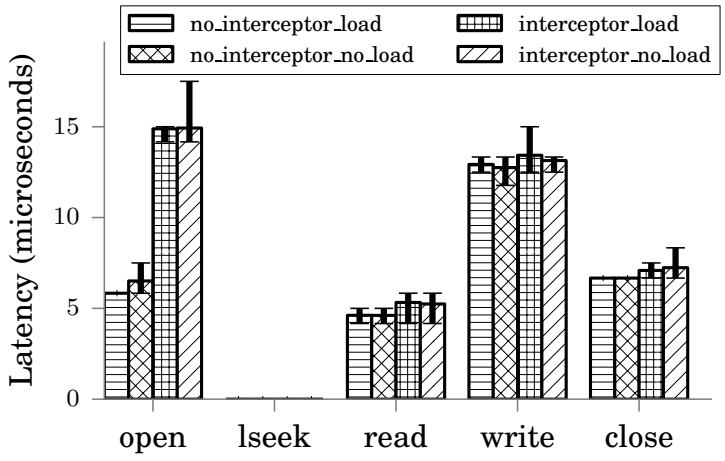
What is the latency and throughput of the system calls we intercept for operations on trusted capsules? We measured the latency and throughput of syscall operations on trusted capsules. For latency measurements, we measured the end-to-end time for a syscall and averaged over 100 runs. For throughput measurements, we randomly read and wrote 4KB of data to a trusted capsule for 10

seconds. We varied the chunk size and data size of our trusted capsule to capture the different effects these factors had on trusted capsule performance at the system call level. For each test trusted capsule, we attached an empty null policy that always evaluated to true. We present our results in Figure 6.6 and 6.7.

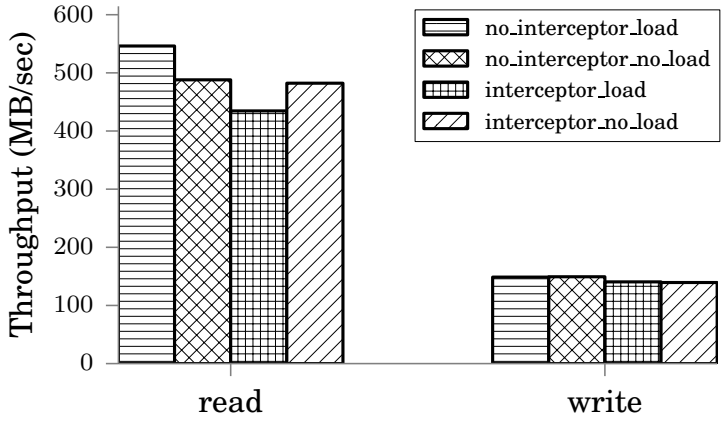
We were able to achieve 1.4MB/s throughput for read and 0.7MB/s throughput for write across all file and chunk sizes. While we initially expected some impact to read and write performance at smaller chunk sizes, our results showed otherwise. We believe this may be due the fact that the cost of extra round trips between normal and secure world to fetch the same quantity of data is not significant overall. We also incur significant costs for the *open* syscall, especially at larger file sizes. While we do not expect the *open* syscall to be called often by applications that interact with a trusted capsule, its poor performance nevertheless requires further optimization. In our current implementation, an *open* syscall makes a pass over the entire data of the trusted capsule to validate the trusted capsule contents against its hash. This results in longer latencies as the file size increases. As can be seen, we were able to obtain reasonable results for *open* calls on file sizes that were less than 100KB. However this latency increases dramatically for larger file sizes.

What is the contribution of various aspects of the trusted capsule monitor to per-operation latency overhead? We evaluated the cost of the (1) normal world data monitor components (interceptor and linuxdriver), (2) world switch, (3) hashing, (4) encryption, (5) secure storage operations, (6) direct file system operations and (7) policy engine initialization. We average our results over 100 iterations of the same system call. Our test capsule had an empty null policy and used 4KB or 1KB chunksizes, and consisted of 1MB or 10KB of data. We show the results for (3)-(7) in Figure 6.8a, 6.9a, and 6.10a. We measured costs of these components end-to-end. For example, a secure storage operation in Figure 6.8a would include the cost of world switches, userspace and kernel space boundary crossings, copying data between layers of the software stack and RPCs. All measurements are based on a 1.2 MHz monotonic counter.

We found accessing secure storage to be an extremely expensive operation, accounting for a large number of cycles. This occurs when policy evaluation needs to access state and on instantiation of the trusted capsule session when the cryptographic information is fetched. Further, the moderate costs associated with direct



(a) The average latency per operation on a regular 1M file in varying system state (load and interceptor) with 95% error bars.



(b) The throughput measured over a ten second period on a regular 1M file in varying system state (load and interceptor).

Figure 6.4: System call interceptor overhead. The system was loaded by accessing 8 different trusted capsules simultaneously

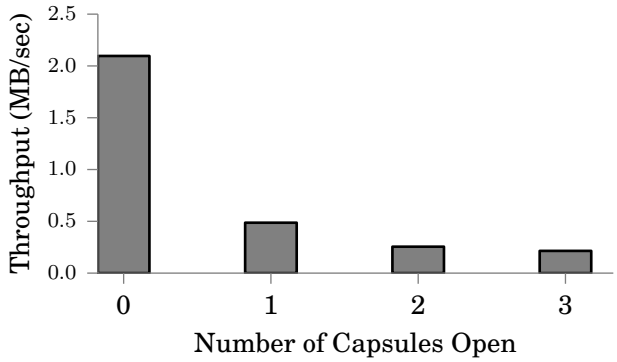
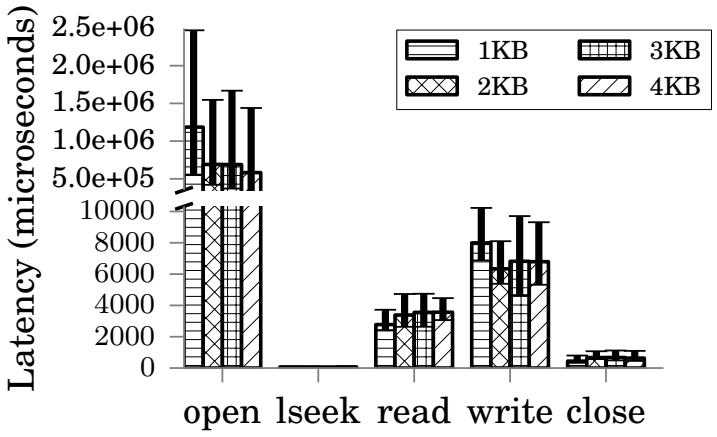


Figure 6.5: The throughput cost to a process for having multiple capsules open while accessing a regular file.

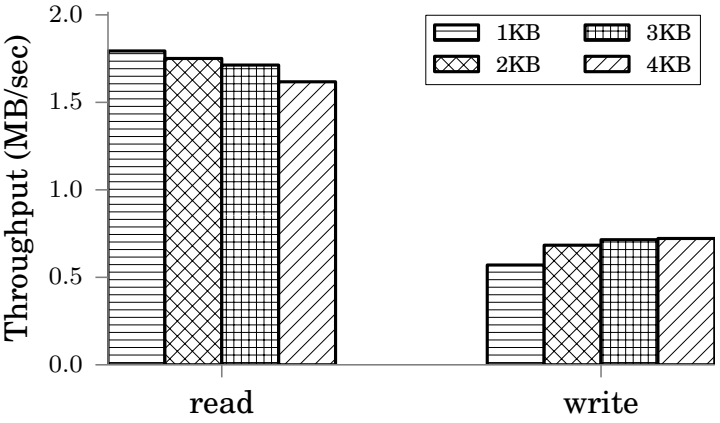
file system operations is in-line with our observation that decreasing chunk size (increasing the number of direct file system operations) did not affect performance extensively. Surprisingly, encryption and hashing was not as a significant component of the overall cost as we thought. Further, initializing the Lua interpreter, the normal world data monitor components (interceptor and linuxdriver) and world switching represented marginal costs. We measured the cost of both the normal world data monitor components and evaluated the null policy to be few dozen cycles and a world switch to cost only a single cycle.

Perhaps the most interesting observation is that all these aspects (1) to (7) combined accounted for only $\sim$30% of total cost of any syscall operation. These operations represent all the transitions that may occur between the trusted application and the remainder of the software stack (normal world components, OP-TEE OS, etc.), except for memory allocations (both between secure and normal world, and between userspace and kernel space within the secure world) and trusted application code. This leads us to believe that a significant portion of the slowdown is due to memory management across the stack and the in-memory copies and loops in the trusted application.

To further evaluate the impact of world switch overhead, we measured the number of world switches for each operation. We broken down the world switch for different purposes. For any operation, we found that it required at most $\sim$100

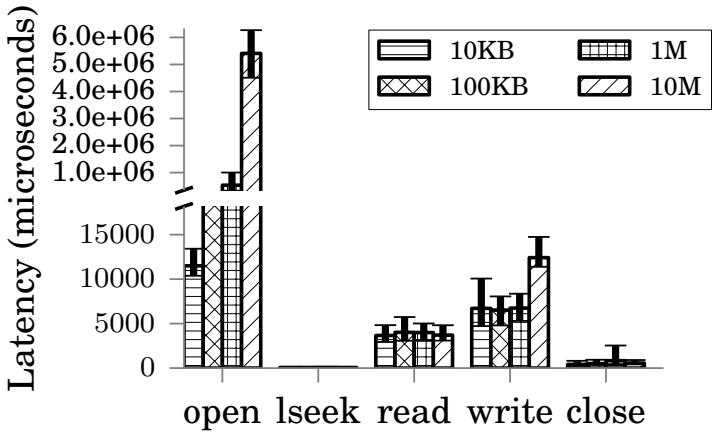


(a) Average latency varied by chunk size. The bars represent 95%.

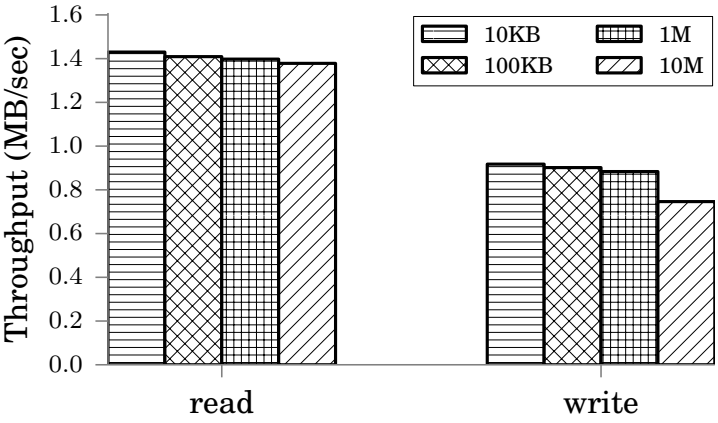


(b) Throughput varied by chunk size.

Figure 6.6: System call performance varied by chunk size. Test trusted capsules had a data size of 1M.



(a) Average latency varied by file size. The bars represent 95%.



(b) Throughput varied by file size.

Figure 6.7: System call performance varied by file size. Test trusted capsules created using 4KB chunk size.

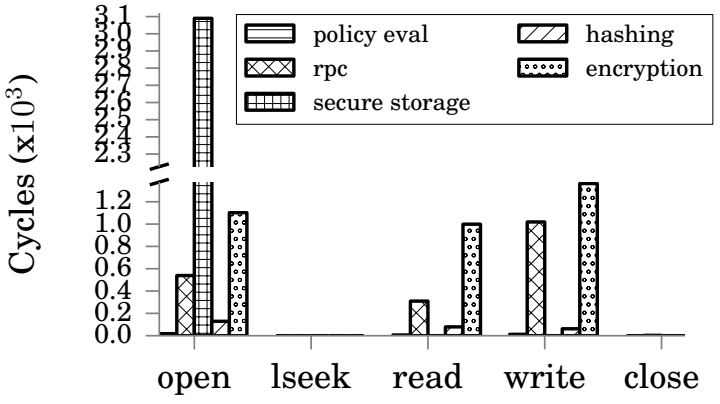
world switches at a cost of 1-2 cycles per world switch. We show the results in Figure 6.8b, 6.9b, and 6.10b. We make an interesting observation that most world switches are used to handle RPCs and their associate shared memory allocations. For example, the read and write operation in Figure 6.9b require more world switches than similar operations in Figure 6.8b due to the greater number of RPC calls required to read at a smaller chunk granularity. Similarly, the open operation in Figure 6.10b require a smaller number of world switches for the similar reason.

6.4 Applications

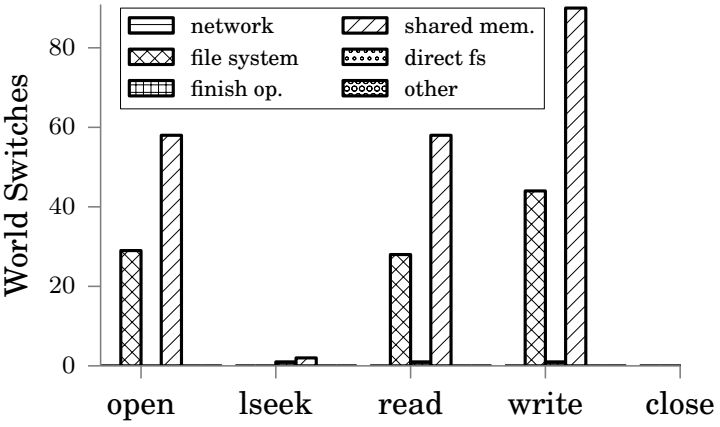
We evaluate the performance of our trusted capsules at the macro-level directly with unmodified applications. We investigated how the overhead incurred by trusted capsules at the system call level impacts performance at the application level. For this purpose we used our use case trusted capsules. We measured the impact under three different conditions: with an empty null policy, with our use case policies, and without trusted capsules (as a baseline). Some of our use case policies, such as those on the PDF and JPEG, required a trusted capsule server in order to fetch remote state and report logged information.

We used a Canon Rebel II DSLR to film certain interactions between the trusted capsule and applications. We then measured the latency between the start of an action (e.g. open a document) and when the action is completed. We filmed at 60 FPS and used the difference in timestamps to calculate the application latency. We present our results in Table 6.3 and provide the raw footage online: https://www.youtube.com/playlist?list=PL5uF0tAnlwkwl5yDQMlJwMOB0aJ4FgbTf.

We note that while overhead improves significantly at the application layer as compared to the system call layer, nevertheless, the cost may be prohibitive depending on policy choice. For various data types, a null policy had much less significant or almost imperceivable impact to usability at the application level. However, our use-case policies, some of which contain expensive policy checks (e.g., access to state storage or going over the network to talk to the policy coordinator) on frequent operations such as read or write, resulted in noticeable performance degradation. For example, our MP4 video played smoothly with a null policy in VLC (which did not interact with the trusted capsule server), but degraded to ex-

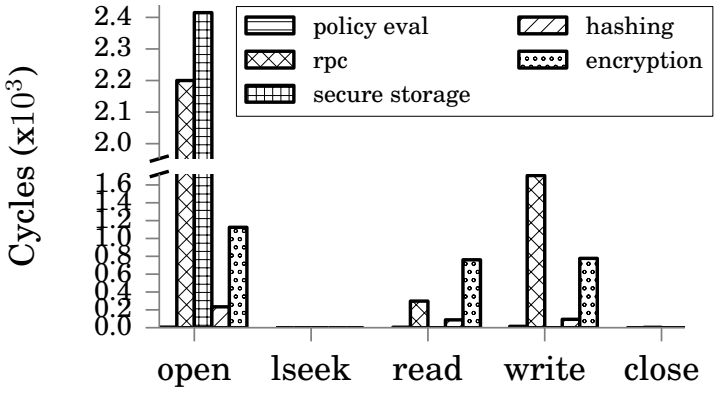


(a) Operations.

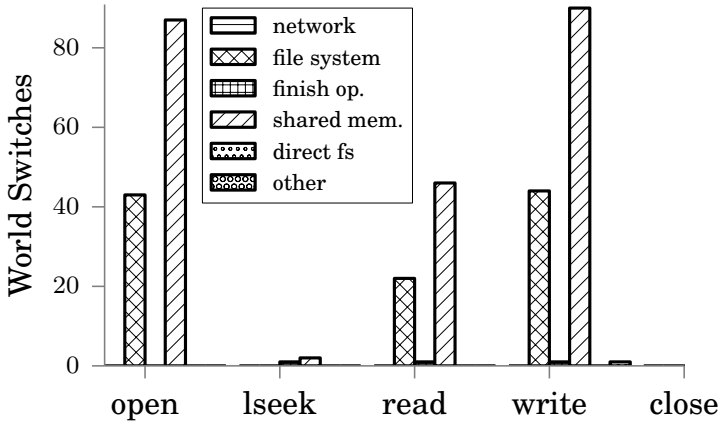


(b) World Switches.

Figure 6.8: Breakdown of number of cycles spent performing specific operations and number of world switches for operations on trusted capsule with 1MB of data at 4KB chunk sizes.

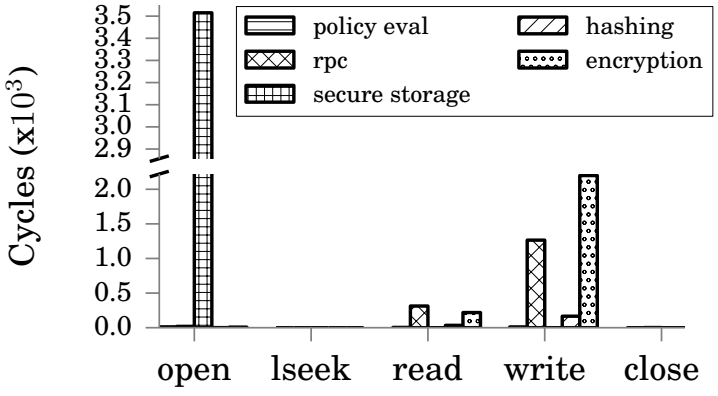


(a) Operations.

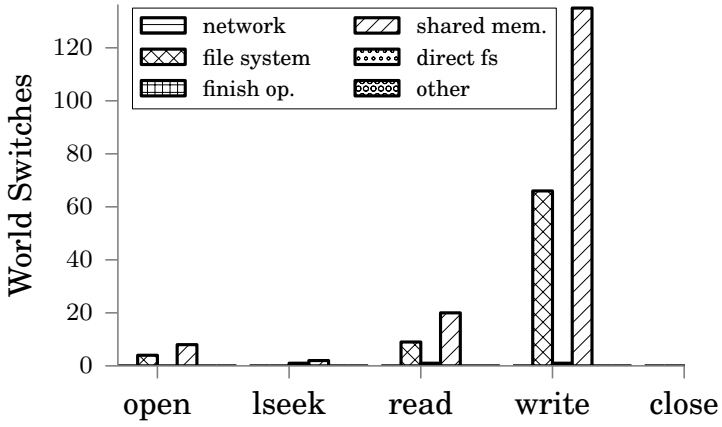


(b) World Switches.

Figure 6.9: Breakdown of number of cycles spent performing specific operations and number of world switches for operations on trusted capsule with 1MB of data at 1KB chunk sizes.



(a) Operations.



(b) World Switches.

Figure 6.10: Breakdown of number of cycles spent performing specific operations and number of world switches for operations on trusted capsule with 1MB of data at 1KB chunk sizes.

treme jitter once we added a policy that reported actions to a policy coordinator and accessed secure storage for every read operation. This effect was particularly acute for the PDF reader, which repeatedly read the data in small chunks frequently and even when the user was idle. Each read by the PDF incurred the cost of a single round-trip to the trusted capsule server, requiring on average 5ms each. We believe that such performance degradations can be mitigated with more efficient policy code, for example policies that run at coarser granularity or use caching to mitigate expensive checks.

Data Type	Application	Interaction	Data	Null	Use-Case
PDF Doc	Evince	Open document	0.87s	3.20s	110.40s
JPEG Image	Gpicview	Open image	0.45s	6.45s	20.23s
		Rotate image	0.23s	0.88s	1.43s
		Save image	0.17s	3.43s	11.38s
MP4 Video	VLC	Video buffer time	3.21s	3.92s	25.72s
FODT	LibreOffice Writer	Open document	1.63s	12.42s	21.67s

Table 6.3: Application level performance. **Data** column represents results gathered from regular data. **Null** column indicates trusted capsule results with NULL policy. **Use-Case** column indicates trusted capsule results with use case policies.

Chapter 7

Related Work

Both past research into tamper-resistant hardware and general data security are relevant to our work. We discuss such work within the context of several generalized categories below.

Mobile Data. Recent academic research has also sought to address the tension between data mobility and security. The vision of policy carrying data was first put forth in [35], but the authors left the implementation as an open question.

One such solution uses attribute-based encryption [39, 41] to establish remote SLAs with an untrusted entity. However ABE encryption is computationally intensive and does not actually enforce the user's policies. Malicious or unintentional exfiltration of the data can still occur without remedy or the user's knowledge. We view our trusted capsule implementation as providing a stronger level of assurance. Access is not granted statically based on the agreement of certain pre-conditions but can be granted dynamically based on current and remote state.

Another class of works, such as P3 [37], are data-type specific and targeted towards untrusted cloud providers. Trusted capsules are data-type agnostic, with the exception of the pre-processor that translates data-specific redaction policies to byte offsets. Further, trusted capsules extend their access boundary to data on remote user devices in addition to the cloud provider. Finally, trusted capsules enable diverse policy capabilities on these remote user devices, such as revocation.

Other solutions such as Ryoan [27] are targeted towards a specific class of applications (e.g., return oriented). In Ryoan, both the application and platform are

untrusted. Ryoan provides enforcement over data derivatives by preventing declassification through tertiary channels. By contrast, trusted capsules target interactive user applications where the data must be declassified to analog holes (e.g., screen), making giving confidentiality guarantees on data derivatives post declassification impossible. Instead, we take a pragmatic approach and enforce advisory policies which can be used to orient behaviour to those desired by the data owner.

Finally, within industry, a recent startup called Sandstorm abstracts data as a *grain* – a package of all the apps, libraries, and configuration files needed to operate on a single piece of data locally within a container. Sandstorm then creates an enclosure around the container and interposes on all operations to enforce the *grain*'s access policies. Unlike trusted capsules, which operates at the granularity of a piece of data, Sandstorm operate at the granularity of an entire software ecosystem for the data.

Hardware-based Security. Many other commercial and research projects have also used tamper-resistant hardware to provide a secure environment [13–16, 20, 23, 30, 33, 34, 40, 42].

In Terra [23], it provides software-based isolation based on building a root-of-trust from tamper-resistant hardware for running applications as isolated VMs. In [40], a .NET runtime was installed in ARM TrustZone to run the security-sensitive parts of an application and protect sensitive information such as passwords, credentials and credit cards. In [20, 30], they created isolated TEEs for executing third-party applications and to store their data. They also provided a mechanism for distributing keys through secure channels into the TEEs of remote devices – an mechanism that can also be adopted for the distribution of trusted capsule keys and credentials. However, all these solutions are meant to strictly provide isolation between the security-sensitive and un-secure components of an application. In trusted capsules, we do not attempt to decompose the application into its secure and un-secure components. Instead, once an application has accessed a trusted capsule, we enforce the trusted capsule's policy by interposing on all future operation by the application. In this way, trusted capsules are compatible with existing applications and can be applied to applications whose security and function cannot be decomposed separately, such as a text editor. More recently, in [15], the authors explored using ARM TrustZone as the TEE to validate the

device and to disable select peripherals upon entry into a secure space such as a federal building or exam center. In [16, 33], tamper-resistant hardware TPMs have been used to provide root-of-trust for distributed systems. TPMs are more limited in their range of capabilities compared to TrustZone, which are capable of general execution and can access main memory. Rather than protecting against malicious or exploitable applications, other works such as [13, 14, 42] focused on using the higher-privileged TEE to protect the host OS from attacks by interposing on all privileged instructions (e.g., pagetable modifications, access to MMU) and redirecting them to the TEE for monitoring. Alternatively, TEEs have also been used to protect the integrity of peripheral devices [34]. In trusted capsules, we use our secure environment to protect both sensitive data and enforce our advisory policies.

Data Confidentiality. Within academia, a well studied approach to providing data confidentiality has been label-based solutions such as DIFC [17, 19, 31, 36, 38, 43, 44]. They use labels to specify access control, capabilities, and authority. These labels are used to track the flow of information at various levels of the software stack. By not allowing data to move to processes that do not have the right labels, DIFC prevents sensitive data from being exfiltrated. DIFC solutions can be enforced statically at compile time, through extension of the programming language [36], or dynamically during execution [19, 31, 43]. Alternatively a DIFC solution can use a combination of both techniques [17, 38]. Depending on implementation, data flow can be tracked at the granularity of address spaces [31], process [19], etc. DIFC solutions operate on the principle of least privilege in an effort to minimize exploitable flaws in applications. It allows a process to gain the minimum privileges it needs in order to perform its execution. In DIFC, labels create a natural ecosystem for composition that allow a process to access multiple pieces of data. Trusted capsules are less composable. If two trusted capsules have contradictory policies, they cannot be accessed by a process at the same time. This may or may not be a desirable property. However, trusted capsules have its advantages. First, DIFC requires customized operating systems or radical modifications to existing applications, which trusted capsules do not. Second, composition is extremely complex as the system administrator must reason about the security lattice between all the processes and data on a system. This is because, in DIFC, ”policy” is not a first-order entity, but implicitly expressed as a combination of the

capabilities, authorities, secrecy and integrity labels assigned to the processes and data on the system. With trusted capsules, policy is a first-order abstraction that singularly embodies all the label, capabilities and privileges needed to access the trusted capsule's data. As a result, trusted capsules do not have a complex security lattice to consider. Second, trusted capsules are backward compatible and do not require extensive modifications across the software stack as opposed to DIFC.

Another popular approach is tainting [21, 22, 26, 45]. It tracks information flow by interposing on the system operations at the instruction-level. These solution can track the flow of information at extremely fine granularity. However they are resource intensive, both in memory and CPU.

A common issues with academic solutions is the lack of compatibility with existing applications. Traditional isolation-based solutions are the only class of practical solutions currently adopted by industry for this reason. These solutions, such as VPN, VMWare Ace [1], Secure Spaces [9] and Hypori [5], attempt to prevent sensitive data from leaving in the first place by enforcing policy at the network boundary between external and internal systems. For example, VPNs enforce policy by acting as the sole gateway to internal systems, while Hypori converts all remote devices into thin clients, while the apps and sensitive data exist as virtual devices on a secure internal server. In these cases, policies that restrict movement of sensitive data can still be defeated by transformations, such as encryption and compression. In addition, some of these solutions incur substantial network cost as they do not support offline operations.

Finally, other work has sought to ensure data confidentiality by enforcing application structures [25, 32], limiting data lifetimes [18, 28] and providing recourse actions such as backtracing intrusions [24, 29].

Chapter 8

Future Work

The future work for the trusted capsule system can be summarized along four axis: bug fixes, optimization, engineering and new research directions.

8.1 Bugs

BUG #1. When a trusted capsule is constructed with chunk size greater than 4KB, the trusted capsule application is unable to read bytes in the range of 3KB→4KB correctly.

BUG #2. The *exit* system call does not decrement the reference count for a trusted capsule session correctly

BUG #3. Unknown interceptor interaction with init and gnome. Removing the interceptor module seems to be buggy. Shuting down with interceptor module inserted is buggy due to some init failure. Gnome cannot start some applications with the interceptor inserted, although you can still launch applications from xterm (e.g., xterm).

BUG #4. Extremely weird problem where *openat* returns -2 instead of -1. This occurs when the interceptor is inserted and the system call is made on a non- trusted capsule file. Some applications fail as they look for -1 for failure and assume -2 is a valid file descriptor which then fails when they read it.

8.2 Optimization

Improving capsule open. We currently verify the hash of the entire trusted capsule on file open. This does not scale with larger file sizes. The original intention was to ensure the integrity of the trusted capsule on open. We disregarded performance as we assumed an application would not need to open a file multiple times. However, our strace results showed that an application repetitively calls the *open* system call on a file during a single access. This expensive verification may be unnecessary as we already verify the integrity of a read and write at a much finer granularity (e.g., just the chunks being read or written to).

A further problem in our current implementation is the fact that we keep the hashes of all the chunks in memory. The original intention was to minimize the number of RPCs we have to make. However, for large files with small chunk sizes, the hash list became a memory bottleneck in the secure world. To enable the trusted capsule monitor to handle larger file sizes, we may need to trade more RPCs for memory space.

Improving capsule read & write. We currently perform read and write on a single chunk at a time. In the future, for optimization purposes, we should look to read and write multiple chunks at a time to reduce the number of RPCs from the secure world. Simultaneously, this would allow us to potentially read more data than what is required and cache the decrypted contents ahead of time. The exact caching and eviction strategy will depend on application access patterns.

8.3 Engineering

Shadow file in normal world. This would enable faster read/write and enable interception of mmap syscalls. But it would move integrity to advisory since there is the possibility a malicious or compromised OS may wait and write to the shadow copy.

Support more system calls. Applications may use other system calls to interact with trusted capsules such as *mmap*, *pipes*, *execv* family and other variants of the system calls that we already intercept (e.g., *pwrite*). Some of these system calls present challenges, such as *mmap* which may specify a write back policy to the normal world OS and proceed to make modifications in-memory. In this case,

we might not intercept a write onto the trusted capsule.

Multi-threaded applications. Our current system call interceptor uses the thread group ID as the process ID. We may need to expand the tuple to also include the process ID *pid* variable inside of the Linux *task_struct* data structure. This would enable us to also handle concurrent operations from applications that are multi-threaded.

Finer-grained locking. Our system call interceptor maintains extremely coarse-grained locks. This should be replaced with finer locks as it is unlikely our trusted capsule monitor will handle more than a few dozen trusted capsules at a time.

Insertion with redactions. Currently our trusted capsule data section is contiguous. We may need to restructure our trusted capsule into redactable and unredacted sections to support writes. The issue that on a contiguous write, an application that reads redacted data may actually overwrite read trusted capsule data with the redaction replacement character. Combined with revocation, such as a policy change, a data owner may then accidentally disclose redacted information if he is unaware that a redacted section has been written over or shifted locally. This is an artifact of our implementation where the data in a trusted capsule is contiguous. A potential solution is reformatting the data into specific sections.

Porting to OP-TEE 2.0. We should shift our build environment to the 64-bit QEMU. Simultaneously we should upgrade our development environment for Hikey and QEMU to use the repo build that is supplied by the current OP-TEE github. This will also move us from OP-TEE 1.0 to OP-TEE 2.0. This may involve re-implementing some of our modifications to the OP-TEE software stack. It may be worth noting that the OP-TEE Linux Driver has been moved into the Linux kernel itself and the kernel API removed. In the future, we may have to call the relevant TrustZone device functions directly from our system call interceptor. Finally, the new Linux build may be much more stable than our pre-Alpha release we currently use. However, it is also more secure, placing the system call table in read-only memory. We will have to temporarily modify some page table bits in order to insert our system call interceptor module.

Robust error handling. We should make error handling more robust across the software stack in both worlds. Currently, we make negligible attempts to ensure smooth operation in the event of unexpected behaviour.

More policy abstractions. To enable policy to express deterministic bounds on policy evaluation, we need two more abstractions for the lua-based policy language. First, the ability to kill a process that has accessed a trusted capsule. Second, the ability to schedule a callback to trusted world after certain amount of time has elapsed.

8.4 Research Directions

Hardening the normal world OS. Several of our trusted capsule monitor components reside in the normal world. We may explore methods to attest the normal world OS and applications to detect compromises that may affect policy enforcement. This follows the path of works in [13, 14, 42], where the authors attempt to harden the normal world OS by basing the detection mechanisms in the secure world. In this case, we may make stronger statements about our advisory policies.

Further, we can explore giving the secure world control of peripheral drivers along a line of work similar to [34]. This would also allow us to make stronger guarantees about policies that require peripheral state from the local device.

De-classification. We make the conservative assumption that once decrypted data is released from the secure world, it becomes declassified. Further, our advisory policies on de-classification currently assume an entire process is tainted for its entire life-time once it has accessed decrypted data. Such an enforcement mechanism does not protect de-classification by a compromised or malicious normal world OS. Therefore, an avenue for future research is whether we can make finer grained and stronger de- classification guarantees. This may involve giving control of peripherals capable of de-classification (e.g., network, block devices) to the secure world and some form of information flow tracking similar to [17, 21, 31, 38] or containerized solutions such as [8, 27].

Chapter 9

Conclusion

Data security on remote devices that the data owner cannot control represent a unique challenge in our data promiscuous world. Systems exchange data indiscriminately and do not offer the data owner any ability to control access policy on remote devices. At best, data is encrypted to prevent declassification.

We introduced graduated access control and realized it using a trusted capsule abstraction and a data monitor that runs inside ARM's TrustZone trusted execution environment. Our solution builds on the file abstraction and does not require any modification to applications, is gradually deployable, and can be ported to other kinds of trusted execution environments.

Bibliography

[1] About VMware ACE. https://www.vmware.com/support/ace/doc/whatsnew_ace.html. Accessed: 2016-11-26. → pages 47

[2] Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware. Accessed: 2016-04-30. → pages 9

[3] Global platform api specifications. http://www.globalplatform.org/. Accessed: 2016-02-15. → pages 10, 11

[4] Lemaker hikey. http://www.securespaces.com/. Accessed: 2016-04-30. → pages 8, 27

[5] Hypori. http://www.hypori.com/. Accessed: 2016-04-30. → pages 2, 47

[6] Images of duchess of cambridge and children stolen in icloud hack. https://www.theguardian.com/lifeandstyle/2016/sep/23/images-of-duchess-of-cambridge-and-children-stolen-in-icloud-hack. Accessed: 2016-09-03. → pages 6

[7] Royals seek prince george paparazzi photo ban after children targeted. http://www.belfasttelegraph.co.uk/news/uk/royals-seek-prince-george-paparazzi-photo-ban-after-children-targeted-31451393.html. Accessed: 2016-09-03. → pages 6

[8] Sandstorm. https://sandstorm.io/. Accessed: 2016-06-30. → pages 2, 51

[9] Secure spaces. http://www.securespaces.com/. Accessed: 2016-04-30. → pages 2, 47

[10] Secure transcript survey us universities use ssn academic transcripts titus. urlhttps://www.privacyrights.org/blog/secure-transcript-survey-us-

universities-use-ssn-academic-transcripts-titus. Accessed: 2016-06-30. → pages 5

[11] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004. → pages 3, 7

[12] R. J. Anderson. A security policy model for clinical information systems. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 30–43. IEEE, 1996. → pages 6

[13] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014. → pages 45, 46, 51

[14] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. Skee: A lightweight secure kernel-level execution environment for arm. 2016. → pages 46, 51

[15] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi. Regulating arm trustzone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 413–425. ACM, 2016. → pages 45

[16] C. Chen, H. Raj, S. Saroiu, and A. Wolman. ctpm: a cloud tpm for cross-device trusted applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014. → pages 45, 46

[17] W. Cheng, D. R. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in aeolus. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 139–151, 2012. → pages 2, 46, 51

[18] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 61–75, 2012. → pages 47

[19] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event

processes in the asbestos operating system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 17–30. ACM, 2005. → pages 2, 46

[20] J.-E. Ekberg, N. Asokan, K. Kostiainen, and A. Rantala. Scheduling execution of credentials in constrained secure environments. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 61–70. ACM, 2008. → pages 45

[21] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014. → pages 2, 47, 51

[22] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley. Towards practical taint tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92*, 2010. → pages 2, 47

[23] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003. → pages 45

[24] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara. The taser intrusion recovery system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 163–176. ACM, 2005. → pages 47

[25] R. Herbster, S. DellaTorre, P. Druschel, and B. Bhattacharjee. Privacy capsules: Preventing information leaks by mobile apps. In *Proc. of MobiSys*, 2016. → pages 47

[26] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 29–41. ACM, 2006. → pages 2, 47

[27] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association. → pages 44, 51

[28] J. Kannan and B.-G. Chun. Making programs forget: Enforcing lifetime for sensitive data. In *HotOS*, 2011. → pages 47

[29] S. T. King and P. M. Chen. Backtracking intrusions. *ACM SIGOPS Operating Systems Review*, 37(5):223–236, 2003. → pages 47

[30] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 104–115. ACM, 2009. → pages 45

[31] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 321–334. ACM, 2007. → pages 2, 46, 51

[32] S. Lee, D. Goel, E. L. Wong, A. Kadav, and M. Dahlin. Privacy preserving collaboration in bring-your-own-apps. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 265–278, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi:10.1145/2987550.2987587. URL http://doi.acm.org/10.1145/2987550.2987587. → pages 47

[33] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009. → pages 45, 46

[34] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 365–378. ACM, 2012. → pages 45, 46, 51

[35] P. Maniatis, D. Akhawe, K. R. Fall, E. Shi, and D. Song. Do you know where your data are? secure data capsules for deployable data protection. In *HotOS*, volume 7, pages 193–205, 2011. → pages 44

[36] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000. → pages 2, 46

[37] M.-R. Ra, R. Govindan, and A. Ortega. P3: Toward privacy-preserving photo sharing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 515–528, 2013. → pages 44

[38] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. *Laminar: practical fine-grained decentralized information flow control*, volume 44. ACM, 2009. → pages 2, 46, 51

[39] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 175–188, Bellevue, WA, 2012. ISBN 978-931971-95-9. → pages 44

[40] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 67–80. ACM, 2014. → pages 45

[41] S. Saroiu, A. Wolman, and S. Agarwal. Policy-carrying data: A privacy abstraction for attaching terms of service to mobile data. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 129–134. ACM, 2015. → pages 44

[42] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM, 2007. → pages 45, 46, 51

[43] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278. USENIX Association, 2006. → pages 2, 46

[44] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *NSDI*, volume 8, pages 293–308, 2008. → pages 2, 46

[45] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. *Neon: system support for derived data management*, volume 45. ACM, 2010. → pages 2, 47