Cross-device Access Control with Trusted Capsules

by

Puneet Mehrotra

B. Engineering, Birla Institue of Technology and Science, 2013

M. Science, Birla Institute of Technology and Science, 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)

October 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Cross-device Access Control with Trusted Capsules

submitted by **Puneet Mehrotra** in partial fulfillment of the requirements for the degree of **Master of Science**
in **Computer Science**.

Examining Committee:

Ivan Beschastnikh, Computer Science
Supervisor

Margo Seltzer, Computer Science
Supervisory Committee Member

Abstract

Users desire control over their data even as they share them across device boundaries. At the moment, they rely on ad-hoc solutions such as sending self-destructible data with ephemeral messaging apps such as SnapChat. We present **Trusted Capsules**, a general cross-device access control abstraction for files. It bundles sensitive files with the policies that govern their accesses into units we call *capsules*. Capsules appear as regular files in the system. When an app opens one, its policy is executed in ARM TrustZone, a hardware-based trusted execution environment, to determine if access should be allowed or denied. As Trusted Capsules is based on a pragmatic threat model, it works with unmodified apps that users have come to rely on, unlike existing work. We show that policies in Trusted Capsules are expressible and that the slowdowns in our approach are limited to the opening and closing of capsules. Once an app opens a capsule, its read throughput of the file is identical to regular non-capsule files.

Lay Summary

People are increasingly using mobile devices to share digital content with each other. While this is convenient and easy to do, they must trust in good faith that the recipient to handle the data as per their wishes. This is not always the case and there are many data leaks that point to this. There is a need to let the users control access to their data even after they have shared it with someone they have no influence over.

We present TrustedCapsules, a system that enables users to define an access policy for their data that is guaranteed to be followed on the recipients device. Our contribution here is this cross-device access control abstraction that works with unmodified applications. We design this system around hardware enabled security guarantees, and evaluate our prototype to show that it imposes a reasonable slowdown when compared to regular file access.

Preface

All work presented henceforth was conducted in the Networks, Systems and Security (NSS) lab in the Department of Computer Science at the University of British Columbia, Vancouver Campus. This thesis is an original, unpublished work by Puneet Mehrotra, written under the supervision of Ivan Beschastnikh. It builds on previous prototypes developed by Peter Feifan Chen and Amanda Levin.

Table of Contents

List of Tables

List of Figures

Acknowledgments

This work was supported by the NSA National Information Assurance Lab and by the Institute of Computing, Information and Cognitive Systems (ICICS) at the University of British Columbia, Vancouver Campus.

I would like to thank Dr. Ivan Beschastnikh for all his help and guidance throughout the entirety of this project and giving me the courage and inspiration to persevere even when I was ready to hang up my boots. I would also like to thank Prof. Margo Seltzer for giving me valuable feedback and agreeing to be my second reader. More than that, I am eternally grateful to Prof. Seltzer for being the beacon of hope during my time at UBC, and for all her advise, kindness, and warmth.

This work would not have been possible had it not been for Dr. Ali Razeen. The time and energy he spent on this project and in guiding me through the maze of academic publishing has made me develop a deeper understanding and appreciation towards the process. His optimism perfectly balances his scepticism (which a security researcher has oodles of) and gave me the much needed confidence to keep calm and carry on. I would also like to thank Eric Semeniuc - undergraduate extraordinaire - for sticking through with the project and executing to perfection the many things I asked. I could not have asked for a better partner.

I would also like to thank all the members of NSS lab who were with me along this journey through graduate school: Amanda Levin (née Carbonari), Clement Fung, Fabian Ruffy, Adam Geller, Vaastav Anand, Surbhi Palande, Nodir Kodirov, and Anthony Mason. I am eternally grateful for all the support and making my time at UBC incredibly fun.

In addition to my friends mentioned above, I would also like to thank all my friends at UBC: Hayley Guillou, Anna Scholtz, Giovanni Viviani, Nico Ritschel,

Aarti Kashyap, Syed Mubashir Iqbal, and many others who deserve to be listed. You have kept me sane and going when my research was not, and helped me keep some form of work-life balance.

Lastly, I must thank my family and friends. Three different time-zones and the great physical distance between us never came in the way of your unconditional and complete support of this path I have embarked upon. I am extremely grateful and blessed to have you all on my side.

Chapter 1

Introduction

Modern mobile devices are highly capable and have enabled users to create and share rich content such as videos, pictures, and documents. However, users often have little control over their shared data. As illustrated in Figure 1.1, a user has full control over her file as long as it stays on her device. She loses this control the moment the file leaves her device boundaries. For example, files backed up to iCloud or Dropbox are vulnerable to the security of those platforms and files shared with other users are vulnerable to their benevolence and their device security policies.

Users today rely on ad-hoc solutions. For example, they might use Cryptomator [3] to encrypt their files before backing them up to the cloud or SnapChat to send self-destructing images that are viewable only for a limited period of time [11]. These apps address particular use-cases with coarse controls and do not provide any general-purpose data protection mechanisms.

Existing work has proposed several solutions to let users retain control over their data as it crosses device boundaries. A current state of the art approach is to use a hardware-based trusted execution environment (TEE) to control accesses to sensitive files. The focus is to ensure that users retain *full* control over their shared data. DroidVault [43], for example, does not allow regular apps running outside the TEE to access the data. Instead, it requires data owners to explicitly write and whitelist the code that is allowed to process sensitive data, and it executes this code within the TEE. We believe that such restrictions make the corresponding systems

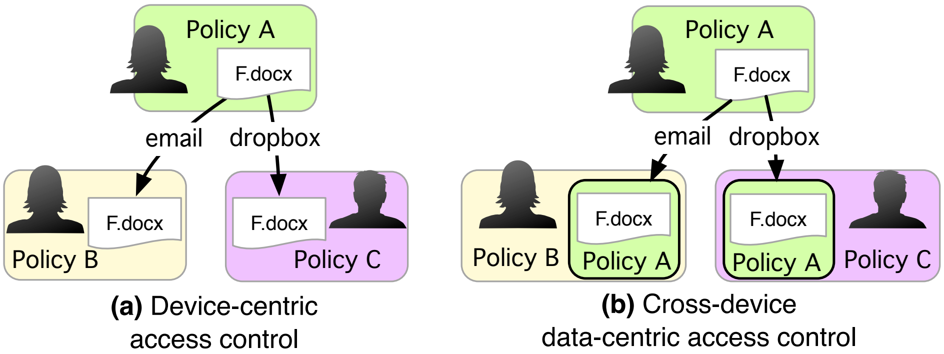

Figure 1.1: (a) Today, a data creator has no control over their data on remote devices: devices enforce local policies on data they receive. We propose (b) cross-platform policies that move with data and are enforced uniformly across devices.

impractical. Users already trust and rely on a variety of apps to create and share content. It is unlikely that users would use a system that does not support their apps.

We present a platform-level file protection mechanism that does not restrict users into using certain apps. To achieve this, we rely on a pragmatic pessimistic-optimistic threat model. In the pessimistic state, we consider the device and apps completely untrusted and rely on a TEE to perform safety checks. When it is considered safe, the system transitions into the optimistic state where we also trust the OS kernel and the app accessing sensitive data. Finally, when the app no longer uses the data, the system switches back into the pessimistic state. We leave a further discussion of our threat model to Section 3.

We contribute **Trusted Capsules**, a data-centric access control abstraction that provides a mechanism to protect against this threat model. It enables users to bundle sensitive files with flexible policies that govern their accesses into encrypted units we call *capsules*. Each capsule appears in the system as a regular file. When an app attempts to open a capsule, the platform evaluates the policy in a TEE. If the the policy allows the access, the capsule's contents are unsealed (decrypted) and provided to the app and are resealed (re-encrypted) when the app later closes the file. In our prototype, we use ARM TrustZone as the TEE and design policies as stateful programs that can base access decisions on information such as location, time, or the number of prior accesses and may, if necessary, modify the data itself (e.g., for redaction).

Our contributions may be summarized as follows:

- A pragmatic access control abstraction for protecting sensitive files across device boundaries that works with existing unmodified apps.
- Using our prototype, we show that our proposed approach imposes slowdowns only when a capsule is being opened or closed (An overhead of 1.96x and 1.67x, respectively, in the absence of any real security policy). Once a capsule file is open, data can be read at a throughput identical to reading regular files.

Chapter 2

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 Trusted Execution Environment (TEE) that is resistant to potential compromise of both applications and OS running on the remote device. We use **ARM TrustZone** technology as our hardware-based **Trusted Execution Environment** and **Linaro OP-TEE** as the operating system that runs in our TEE. Within this TEE, we handle sensitive cryptographic operations, perform policy evaluation, securely store policy state, and establish a secure channel to the remote policy coordinator server.

In the following, we provide a brief overview of the properties of TrustZone and OP-TEE.

2.1 TrustZone

ARM TrustZone [13] is widely available on current commodity ARM processors. A TrustZone enabled processor maps to two virtual processors that that execute in a time-sliced fashion, context switching between each other through a special core mode called the “monitor mode”. We call a virtual processor that is running in the TEE to be running in **Secure World**. A virtual processor that runs the regular operating system is said to be running in **Normal World**. The monitor mode software acts as a robust gatekeeper to manage the physical processor's context switch between these virtual processor modes by means of interrupt handling. These modes of operation of a physical processor and the transitions between them are visualized in Figure 2.1.

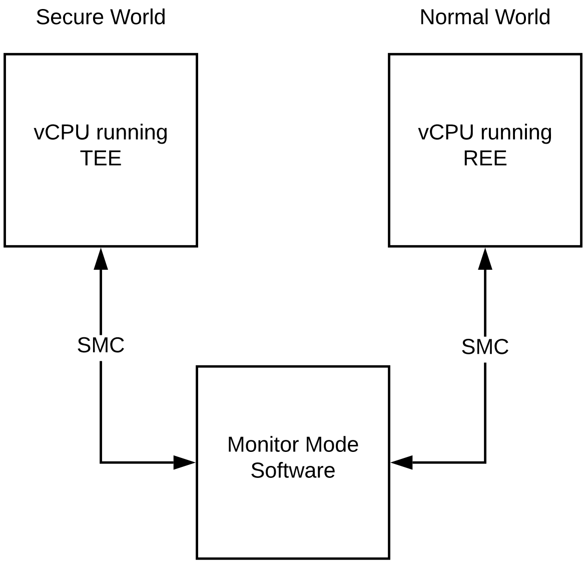

Figure 2.1: The Physical CPU maps to two virtual CPUs - one running the TEE (secure world) and the other running the rich OS (normal world). The context transitions between these are handled by the `smc` instruction that gets serviced by the monitor mode software, which runs at the highest level of privilege.

One way in which world switch can be triggered by the software is by executing a dedicated instruction, the Secure Monitor Call (SMC) instruction. The software that runs in the monitor mode is defined by the implementation of the chipset, but it generally saves the state of the current world and restores the state of the world being switched to. Once the restoration is complete, the monitor performs a return-from-exception call to restart the processing in the restored world.

A TrustZone enabled processor implements three sets of interrupt vectors - one for each world, and one for the monitor mode. The locations of these tables are programmable and can be modified by changing the appropriate Vector Base

Address Register (VBAR). This can be used to control where the *smc* instruction will trap to. VBARs are only accessible in privileged modes. The secure copy of the VBAR holds the vector base address for the Secure State and the non-secure copy of the VBAR holds the vector base address for the Non-secure state.

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**.[1]

This security guarantee for register access is maintained by tracking the execution mode of a processor. The monitor mode software executes in secure world. The world in which a processor is currently executing is indicated by the NS-bit in the Secure Configuration Register (SCR) in the system control coprocessor (CP15). When in monitor mode the processor is running in Secure World, regardless of the value of the NS-bit, but operations on the banked CP-15 registers will access the Normal World copies if the SCR NS-bit is set to 1.

For memory, the secure world provides such a guarantee by either taking exclusive control of on-chip memory such as secure SRAM [6] 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 (Fast Interrupt Request) and IRQ (Interrupt Request). 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. This is done by propagating the NS-bit over the system bus.

[1] TEEs cross-world ability to manipulate the memory and registers are grounds for Spectre[36] and Meltdown[45] type bugs. This has been acknowledged by Linaro and ARM and they have published page table isolation patches to fix these[32].

Exception Level	**Secure World**	**Normal World**
EL0	Trusted Application	Application
EL1	Secure OS	Normal OS
EL2	-	Hypervisor
EL3	Secure Monitor	

Table 2.1: The privilege levels at which the various components of the TrustZone based system run. The Secure Monitor runs at the highest privilege.

The secure monitor operates at Exception Level(EL) 3 - a higher privilege level than both the application (trusted or regular) and the operating system (secure or normal). The Secure OS and the Normal OS operate at EL1, while the Applications - both in secure and normal world - run at EL0. This is shown in Table 2.1.

2.2 Linaro OP-TEE

Linaro OP-TEE is an open-source operating system that has been designed for ARM TrustZone. Linaro is part of the industry consortium called Global Platform, which leads efforts in standardizing APIs exposed by different TEE OS vendors. OP-TEE is the OS and related firmware that incorporates this standardized API and is ported across several hardware vendors.

OP-TEE is the secure OS for executing trusted applications and is composed of:

1. A low-level secure monitor for world-switching (ARM Trusted Firmware). This is where the SMC implementation resides.
2. A TrustZone driver (OP-TEE Linux driver) which is used to access the TEE services from the normal world, and
3. OP-TEE Supplicant which runs in normal world user space as a single threaded application and is responsible for accessing services expected by the TEE-OS.

The remaining discussion in this section is based arouind the HiKey system-on-chip (SoC) with Debian Linux as the normal world OS. This is configuration

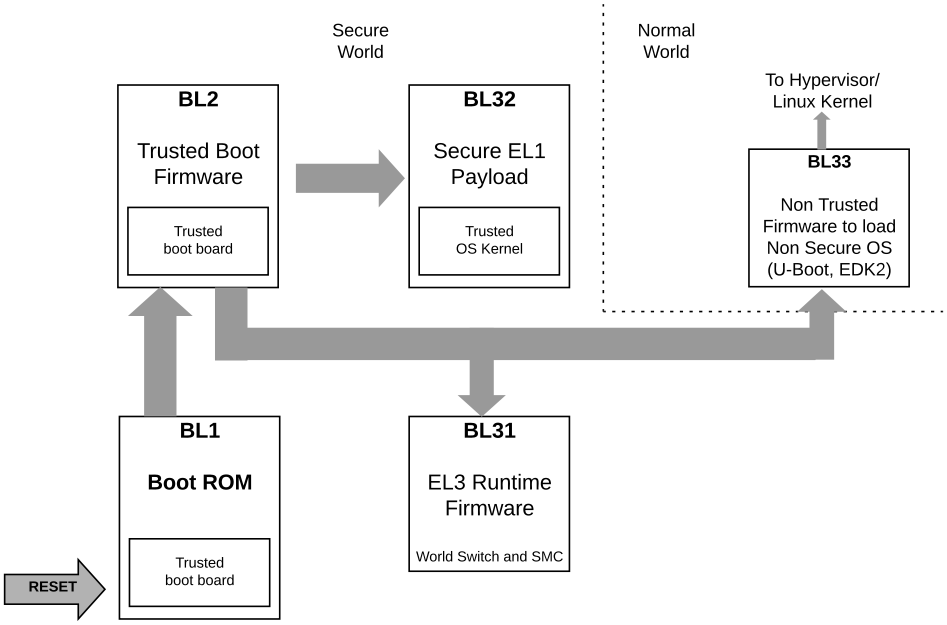

Figure 2.2: ARM TrustZone Boot Sequence.

we used for this project.

2.2.1 ARM Trusted Firmware

ARM Trusted Firmware (ATF) [2] provides a reference implementation of the Secure Monitor and the varoius Arm interface standards around system control and management, secure boot conventions, and interrupt management interface. It is the critical piece in booting the secure environment, a process which is outlined in Figure 2.2. There are three bootloaders involved in the process, and each is responsible for initializing the image for the next level in the process. BL2 loads all images in the third level of initialization. A root-of-trust can be built by having each stage attest the image of the next.

2.2.2 OP-TEE OS

OP-TEE OS is a small operating system that has been designed to run in the TrustZone backed TEE. It supports multi-threading and memory management, and some peripheral control over GPIO pins. Despite being a multi-threaded multi-core operating system, OP-TEE does not have a scheduler. OP-TEE does not differentiate between single core and multicore hardware - which core it runs on depends on which core was in use in the normal world user-space when the SMC call was initiated. On receiving the SMC call, the first trap occurs to the *cpu_on_handler()* call on a fixed core (usually core 0), and it finishes with the SMC switching back to ARM-TF (EL3) and then the dispatcher does the world change to another core.

Communication between the normal world and secure world occurs over buffers that are allocated by the normal world but are managed by the secure world. These buffers are used to pass information to and from the secure world for tasks such as TEE function invocation and Normal world filesystem access from within the TEE.

OP-TEE OS provides APIs that can be used to construct user space trusted applications running in secure world (EL0 in secure world). OP-TEE OS applications conform to the GlobalPlatform Internal API [5] where each trusted application must implement a set of well-defined functions as entry-points. These trusted applications run in the secure world user space (Secure EL0 in Table 2.1). Trusted applications can have a single-instance running in the secure world. Trusted Applications can, however, have multiple sessions active with the normal world. Client applications in the normal world invoke these trusted applications through a similar set of GlobalPlatform Client APIs. The flow of one such API call - to load a trusted application and establish a session with the trusted application is shown in Figure 2.3.

2.2.3 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 Linux TEE driver.

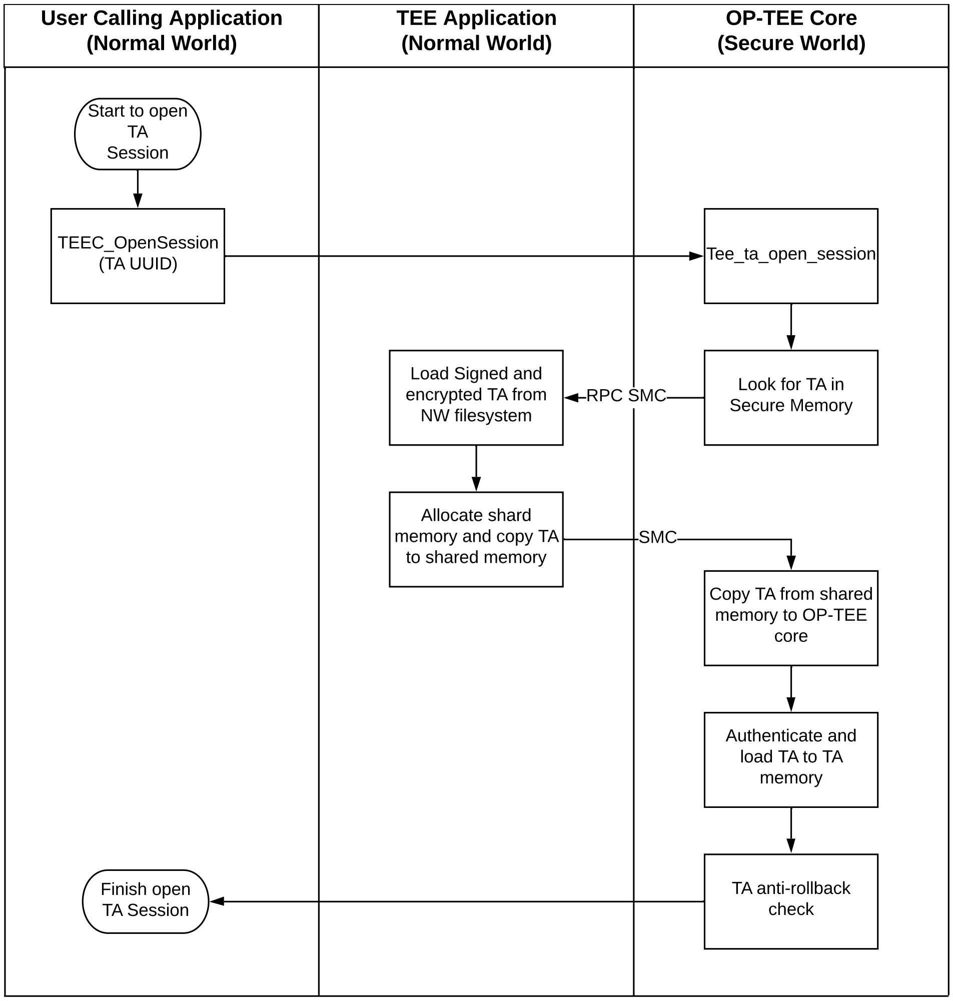
User Calling Application
(Normal World)
TEE Application
(Normal World)
OP-TEE Core
(Secure World)
Start to open TA Session
TEEC_OpenSession (TA UUID)
Tee_ta_open_session
Look for TA in Secure Memory
RPC SMC
Load Signed and encrypted TA from NW filesystem
Allocate shard memory and copy TA to shared memory
SMC
Copy TA from shared memory to OP-TEE core
Authenticate and load TA to TA memory
TA anti-rollback check
Finish open TA Session

Figure 2.3: OP-TEE API calls to open A TA session

Chapter 3

Threat Model

3.1 Contextual Theat Model

Trusted Capsules use a threat model that changes based on the context in which the application is executing. As illustrated in Figure 3.1, this model has two states, *pessimistic* and *optimistic*, and there is a transition between the two states depending on the context. We assume that device owners have full control over the software stack running in the normal world but may not modify the stack running within TrustZone.

The system begins in the *pessimistic* state when the user first receives a capsule, which is encrypted data bundled with a policy that governs its access. In this state, the TCB consists solely of ARM TrustZone and the secure monitor; the OP-TEE OS running in TrustZone; the Trusted Capsules data monitor that runs in OP-TEE OS. All code running outside of TrustZone (i.e., the normal world kernel and apps) are considered untrusted. In this state, we guarantee that the capsule's decrypted contents are not available and that it is safe from attempts to either exfiltrate or modify its data or policies. When the user opens the capsule with an app (which will use the `open()` syscall), the policy embedded in the capsule is executed by the Trusted Capsules data monitor. If the policy denies access to the file, the system remains in the pessimistic state (and the app's call to `open()` will fail).

If access is allowed, the system transitions to the *optimistic* state where the decrypted capsule data is given to the app. The TCB in this state expands to in-

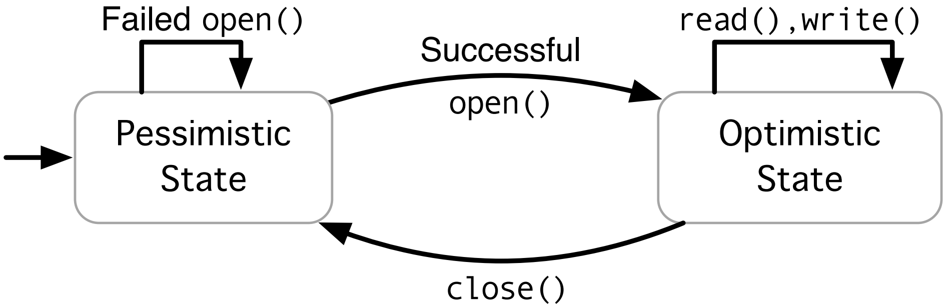

Figure 3.1: A finite-state machine view of Trusted Capsule's threat model. Each capsule on a device begins in the pessimistic state. A successful transition from the pessimistic to optimistic state means an app on the device tried to open the capsule and the capsule's policy authorized the access. Only that app process is allowed to access that file in the optimistic state. When the the process closes the file, the system transitions back to the pessimistic state.

clude the normal world kernel and the app that opened the file. Only that app is authorized to access the file and we rely on the process isolation mechanisms in the normal world kernel to prevent other unauthorized apps from accessing the decrypted data. When the app closes the file (with the `close()` syscall), the capsule is re-sealed and the system transitions back to the pessimistic state. If the app modifies the file before closing it, the changes are saved only if allowed by the capsule policy. Otherwise, they are discarded and the capsule is resealed with the original capsule data. Any session data in TrustZone is also discarded.

We consider side-channel and analog attacks out-of-scope. We can not control the application from transmitting the capsule's decrypted contents during the optimistic state of operation.

3.2 Discussion

Table 3.1 summarizes the protections Trusted Capsules offers depending on the adversary's capabilities and the state of the system. Consider the scenario when Alvin sends a capsule with a photo to Barbara's smartphone with a policy that requires Barbara to authenticate herself before she can view the photo. In the worst

State Adversary	Pessimistic	Optimistic
Weak	✓	✓
Strong	✓	✗

Table 3.1: An enumeration of the possible system state and adversary type combinations. The ✓ and ✗ symbols indicate whether or not Trusted Capsules prevents data exfiltration in the corresponding scenario. Note that the adversary here is not authorized to directly open a capsule on the device.

case, Barbara herself is an adversary interested in leaking the photo. There are no mechanisms in Trusted Capsules preventing Barbara from doing so; the best that can be done is for Alvin to be sure he trusts Barbara before he authorizes her to view the photo.

Consider instead the situation where Barbara is trustworthy but her smartphone is sometimes accessible by Charlie, an adversary who is secretly interested in viewing Alvin's photo. Charlie aims to modify the state of the smartphone so that when Barbara subsequently regains control of her phone and opens Alvin's capsule, Charlie surreptitiously receives a copy of the photo.

If the system is in the pessimistic state, there is no way for Charlie to view the photo because the capsule is sealed and encrypted. In the optimistic state, whether Charlie can exfiltrate the photo depends on whether he is a *weak* or a *strong* adversary. A weak adversary is one who is not technically inclined and hence may not do much more than install new apps from the smartphone app store. In this event, when Barbara opens the capsule and the system switches to the optimistic state, Trusted Capsules relies on the kernel's app isolation mechanisms to prevent other unauthorized apps Charlie might have installed from accessing the decrypted capsule data.

On the other hand, if Charlie is a strong adversary, then he may use a variety of techniques such as kernel modifications to access the photo in the optimistic state. While Trusted Capsules does not protect against this scenario at the moment, it may be mitigated by having the policy reason about the normal world software stack before opening the capsule. We leave an investigation of this strategy to future work. Finally, note that regardless of the system state and adversary type, an adversary may not alter the policy embedded in a capsule and it never leaves the TrustZone environment.

Chapter 4

Trusted Capsules

In this section, we describe the components of our system in more detail and describe how they work together. A capsule (Section 4.1) is the data encrypted together with its access policy. This access policy is written in Lua and uses Trusted Application's functionality defined by the Policy API (Section 4.2). The handover of the data from the normal world to the secure world is handled by a FUSE based data monitor (Section 4.3).

4.1 Capsules

A capsule consists of data and an access policy for the data, both encapsulated into a single encrypted file. Figure 4.1 illustrates the format of a capsule. A capsule has an unencrypted header segment (Shown in blue in Figure 4.1) followed by an encrypted data block (Shown in pink in Figure 4.1). The header identifies the file as a capsule and contains integrity metadata used by the data monitor:

1. **Trusted Capsule Identifier**: This is used to identify that the file being accessed is a capsule. This identifier spells "TRUSTEDCAP" in plaintext.

2. **Capsule UUID**: This is a unique identifier that gets assigned to the capsule at the time of creation. This is used to find the decryption keys in TrustZone.

3. **Capsule Size**: This field stores the size of the capsule in bytes. This is used to create communication buffers to pass the contents to the secure world.

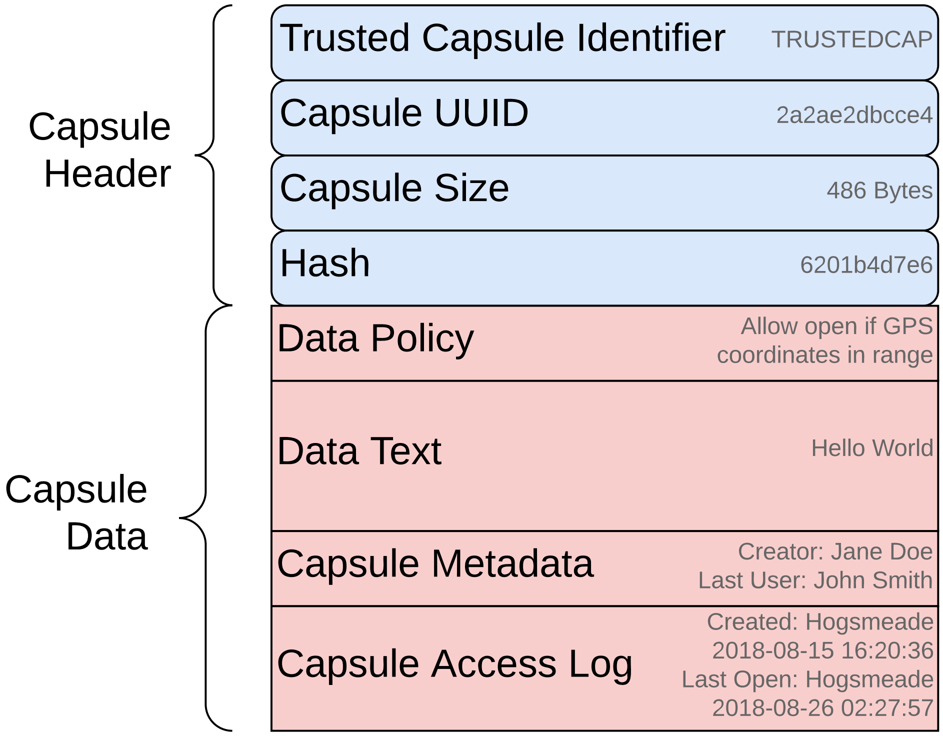

Figure 4.1: Trusted capsule layout.

4. **Hash**: This field stores the hash of the data block of the capsule.

The data block contains the following:

1. **Data Policy**: This Lua Policy script that is run by the Policy Engine in TEE. This is written in accordance with the Policy API.

2. **Data Text**: These are the exact contents of the file being encrypted.

3. **Capsule Metadata**: This section is to hold any Key:Value based metadata that the Policy Engine might need to evaluate the policy.

4. **Capsule Access Log**: This section holds the latest accesses to the capsule. Every time a capsule is accessed, an entry gets appended here.

The cryptographic keys required to decrypt capsules are securely loaded into a

	Description
Open-Only	
`redact`(*start, end, replaceBytes*)	Replace byte range [start, end] of trusted capsule data with bytes replaceBytes.
Close-Only	
`readNewCapsuleData`(*offset, length*)	Return length bytes from offset of new trusted capsule data.
`newCapsuleLength`()	Return the length of new trusted capsule data.
Shared	
`getState`(*key, where*)	Get state mapped to key from where.
`setState`(*key, val, where*)	Set state mapped to key to val in where.
`getLocation`(*where*)	Get location of device from where.
`getTime`(*where*)	Get current time from where.
`readOriginalCapsuleData`(*offset, length*)	Return length bytes from offset of original trusted capsule data.
`originalCapsuleLength`()	Return the length of original trusted capsule data.
`deleteCapsule`()	Delete the trusted capsule.
`updatePolicy`()	Check for policy update with trusted capsule server.
`appendToBlacklist`(*key, where*)	Append key to blacklist of where - used by log to prune states in where.
`removeFromBlacklist`(*key, where*)	Remove key from blacklist of where.

Table 4.1: The Lua-based API that policies in Trusted Capsules may use

secure storage area accessible only by the TEE. The protocol to securely load keys into the TEE is described in Chapter 5.

4.2 Policy API

In Trusted Capsules, policies are written in the Lua programming language and have one simple requirement: they must implement an `evaluate_policy`(*op*) function that is called when the capsule is being opened or closed; the *op* argument distinguishes between the two. There is a basic sanity check in the trusted application to ensure that the operation is valid - for example, the calling application cannot request a `close()` on a capsule that was never opened. In either case, the function has to return a boolean value that is interpreted differently depending on the operation. If it returns `true` on a capsule open, the data is decrypted and given to the normal world app. Otherwise, access is denied. On a capsule close, returning `true` means file modifications by the normal world app will be kept while `false` means they will be discarded. Policies may also use the Trusted Capsules API listed in Table 4.1 to easily perform common operations:

Storing state: Policies may store and retrieve arbitrary state using the state-oriented APIs such as `getState` and `setState`. When using such methods, the policy must specify *where* the state is to be kept. A policy may securely store state in the metadata space within its capsule, in external secure storage, or at a remote server. If a policy communicates with a remote server, the networking stack in the normal world kernel is used to initiate the connection. However, as the OP-TEE OS includes the mbed TLS library [8], it is possible to safely make an HTTPS connection from the secure world without trusting the normal world.

Ensuring data integrity: Our Lua policy provides APIs to retrieve the original trusted capsule data at file open (read) and the new trusted capsule data at file close (write). Using these APIs, data owners can express policies that, for example, protect specific data regions from being overwritten.

Redaction: Selective policy-based disclosure of trusted capsule contents is a key feature of trusted capsules. Using our byte-oriented redaction API, data owners can express arbitrary data transformations on regions of the data based on the environment and the state of the device *prior to* disclosing information to the normal world. Examples of data transformations include removing sensitive texts or blurring images.

Revocation: A policy can specify revocation in two ways. First, we provide

APIs to allow policies to self-delete a trusted capsule. When the *deleteCapsule* API is called, we overwrite the trusted capsule file with zeros[1]. 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 remote capsule server. In this scenario, the policy specifies a condition under which *updatePolicy* is called. If a new policy exists at the trusted capsule server, it is downloaded by the trusted world and replaces the prior policy. Policy changes are temporary as the owner could always change the policy back.

Logging: We extended the Lua language with the ability to report information to the remote capsule server. To enable logging on open and close, *log_open* and *log_close* flags must be set to true, respectively. By default, the Lua sandbox will report the location, identity, time, and the operation. Additional local or capsule state information is also logged, unless otherwise specified by the APIs *appendToBlacklist* and *removeFromBlacklist*. The logs are written into the LOG section of the trusted capsule. If the section runs out of space, the logs are flushed to the remote server and then overwritten.

4.3 Data monitor

In Figure 4.2, we illustrate the different components of the data monitor in our system and in Figure 4.3, we show a detailed data flow between them when an application opens a capsule. These components may be broadly classified into (1) framework code that runs in the normal world OS, and (2) a policy execution engine in the secure world. Next, we discuss each component in detail while referring to the data flow in Figure 4.3.

Normal world framework: We implemented a passthrough FUSE filesystem in the normal world and expose it as a separate mount point. When an application opens files located on this mount point, our framework will interpose on the application's `open` syscall. It will check the header of the file to identify if it is a capsule. If it is a regular file, it will just load the raw file from the underlying file system and return it to the app.

[1] This is because the Linux OS does not delete the file until the file's reference count becomes zero

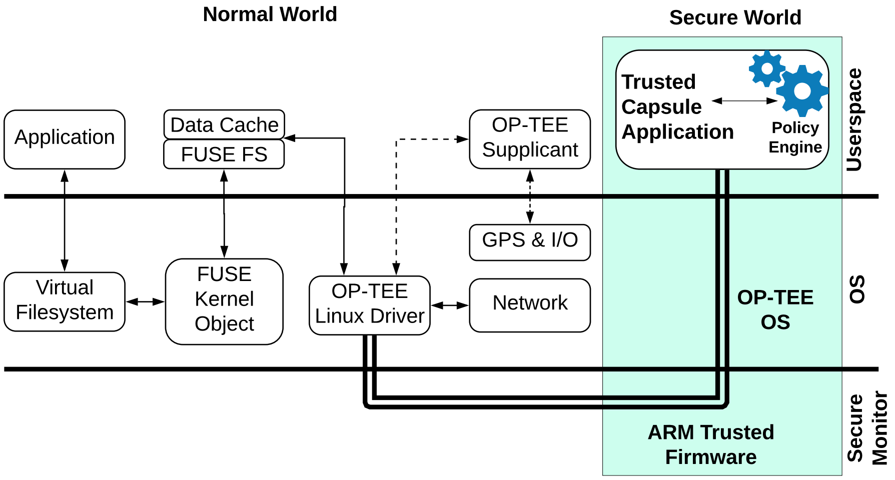

Figure 4.2: Trusted capsule data monitor design. Application system calls to the filesystem for accessing trusted capsules are intercepted and forwarded to the trusted capsule application through the FUSE filesystem and OP-TEE Linux Driver. The secure world trusted capsule applications access peripheral I/O through RPC calls to the OP-TEE Supplicant via the OP-TEE Linux Driver.

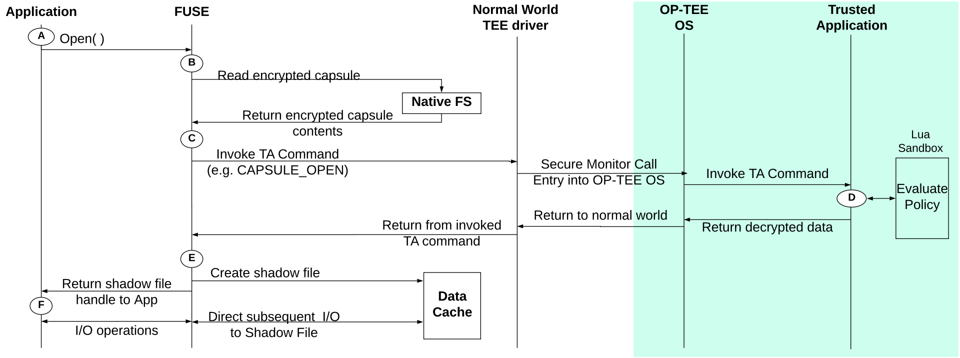

Figure 4.3: Trusted capsule monitor operation (shaded region operates in the secure world). **A.** Application *open* system call is intercepted. **B, C.** FUSE identifies if a file is a capsule, and if so, invokes an RPC into the secure world to decrypt the capsule. **D.** The trusted capsule application (TA) evaluates the *open* policy. **E.** FUSE writes the decrypted contents to a shadow file **F.** The application is returned a filehandle to the shadow file, and all subsequent I/O requests are directed to the shadow file.

If it is a capsule, the file contents are copied into a memory buffer. FUSE then shares this buffer with the policy execution engine running inside the secure world and invokes the engine's *decrypt* function (*A*-*C* in Figure 4.3). If the policy authorizes the access, the policy engine will return the decrypted contents of the capsule and FUSE will save them into a shadow file (*E*). It will subsequently return a handle to this shadow file to the application (*F*). Hence, all reads and writes to the capsule by the application will be transparently redirected to the shadow file.

When the application closes the capsule, FUSE copies the shadow file back into a shared buffer, sends it to the policy execution engine, and invokes the *encrypt* operation. This returns the reconstructed capsule, with the updated policy metadata and data contents (as authorized by the policy), which is then written in place of the original capsule file.

Our framework prevents multiple applications from concurrently opening the same capsule. This simplifies the design of our data monitor as we do not have to reason about multiple-reader/multiple-writer type problems when saving a capsule. An application may, however, have multiple capsules open.

Policy execution engine: We implemented a Trusted Application (TA) that runs in the secure world. It contains a Lua interpreter, to execute a capsule's policies, and it is responsible for maintaining the runtime session state associated with a capsule (e.g., cryptographic keys) and updating the capsule metadata.

When a *decrypt* operation is received from the normal world (because a normal world application used the `open` syscall on a capsule), a new instance of the trusted application is started. It (1) loads the capsule, (2) loads the cryptographic keys for the capsule, (3) executes the policy, and (4) returns the decrypted capsule data if authorized by the policy. During policy evaluation, it may communicate to a remote server directly from the secure world.

On an *encrypt* operation (which is initiated because a normal world application used the `close` syscall to close a capsule), the TA evaluates the policy and provides it the opportunity to allow or deny modifications to the capsule data. Next, it updates the metadata, produces a new capsule file with updated contents in the data block, and updates the integrity metadata in the header. Finally, the reconstructed capsule is given to the normal world for storage and subsequent use.

We use OP-TEE OS native secure storage capability to store our cryptographic keys and persistent trusted capsule states. The 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 capsules. In contrast, each capsule gets its own secure state file. State files can only be opened by a single trusted capsule application at a time. This is enforced through the OP-TEE OS. In this way, we enforce a single trusted capsule instance at a time per capsule.

4.4 Security analysis

We consider two important security aspects of the Trusted Capsules data monitor.

Trusted Capsules: Operations on the trusted capsule are performed by the trusted application in the secure world. We isolate each trusted capsule by having separate instances of the trusted application handle each capsule and by relying on OP-TEE OS to isolate each trusted application instance. Our system stores persistent state associated with capsules (such as cryptographic keys) in the secure storage functionality available in OP-TEE.

Given our use of TrustZone, the confidentiality and integrity of the capsule data is protected against compromises of the normal world OS, particularly in the pessimistic state. A compromised normal world OS may corrupt a capsule, but that corruption will be detected during decryption. In the worst case, a compromised OS may leak the data of capsules that are open during the compromise.

Policy Evaluation: To account for malicious policies, we made several changes to the Lua interpreter to make it a sandbox. We disabled any Lua library that allows the interpreter to interact with external systems (e.g., I/O, packages, debug, and OS). We also extended the interpreter to prevent policies from (1) interacting with any files other than the capsule, (2) from accessing keys associated with other capsules, and (3) reading unauthorized device peripherals. A malicious policy may attempt denial-of-service attacks such as infinite loops. However, these may be addressed even by the normal world, by canceling an *encrypt* or *decrypt* commands that do not complete after some time.

Chapter 5

Device Registration and Key Distribution

There is also a need to register the devices on which the capsule can be accessed and created and simultaneously make known the users who are using these devices. This <`user,device RSA pubkey, approved capsules` > relationship is maintained on the remote server, and is queried when a user requests decryption of a capsule for which she does not have the decryption key.

This access control relationship is developed over two steps - a user needs to register his/her device and the capsule creator needs to create a list of users who are approved to receive the decryption keys. We take a look at both of these steps and then inspect the protocol followed to resolve a decryption request.

5.1 Registering a Capsule Recipient

The process to register a user as a capsule recipient is outlined in figure 5.1. The user initiates a `register` call to the secure world with the email address they use to receive the capsule. The secure world at this point, looks up the secure storage to identify if it has a RSA public/private key pair saved. If it does not, the RSA key pair is generated.On receipt of the `register` request, the secure world handler initiates a TCP request to the remote server and passes the email address and the RSA public key it fetched (or generated).

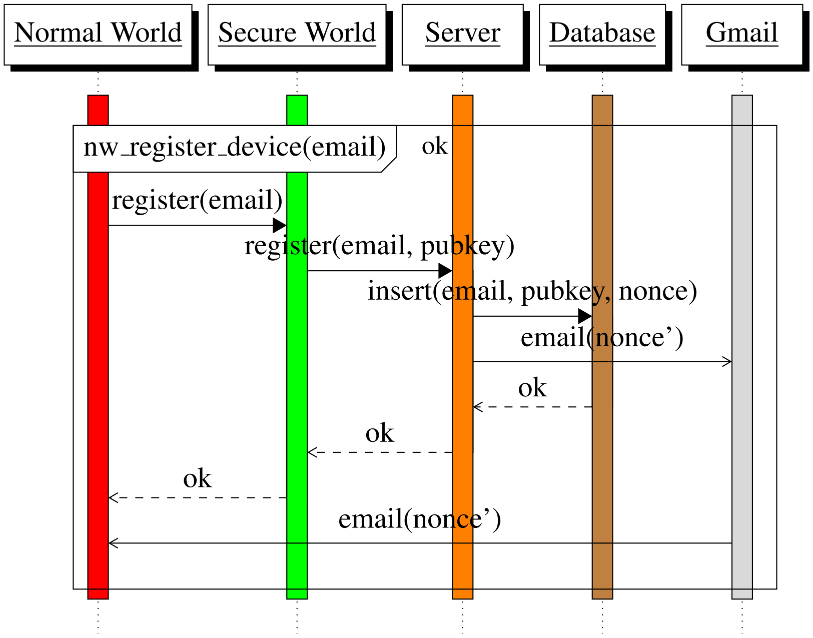

Figure 5.1: Registration as Recipient - intiating the registration process

On the receipt of this request, the remote server generates a nonce for this request and inserts the `<email, device RSA pubkey, nonce>` tuple into it's database. Using the received RSA public key, the remote server encypts the generated nonce and sends an email to the users email address. This marks the first step of the registration process.

When the user receives the email with the encrypted nonce, the second part of the registration process can commence. This part of the process is used to validate that it was indeed the user who sent the registration request and establishes the `<email,device RSA pubkey>` relationship. This process is detailed in figure 5.2.

To begin the validation request, the user passes the received encrypted nonce and the email address to the secure world using the `verify` request. The secure world decrypts the nonce using the private RSA key that is held in the secure storage. Once the nonce has been decrypted, the secure world initiates a TCP connection to the remote server and passes the nonce, email address, and the device RSA public key for verifcation.

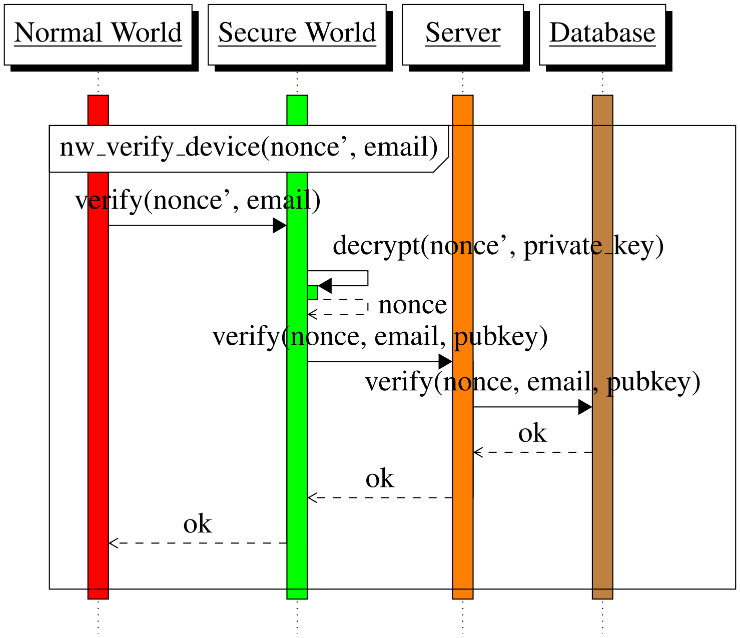
Normal World
Secure World
Server
Database
nw_verify_device(nonce’, email)
verify(nonce’, email)
decrypt(nonce’, private_key)
nonce
verify(nonce, email, pubkey)
verify(nonce, email, pubkey)
ok
ok
ok

Figure 5.2: Registration as Recipient - validation of the request

The remote server verifies that the email id and the nonce it had saved in the database while creating the verification request. If the nonce, email, and the public key all match, the tuple is persisted and an OK status is returned to the user.

At the end of this process, the identity of the user is tied to the email address and the device pair. This process can be repeated on any other devices a user owns. The `<user,device RSA pubkey>` relation is a unique key that is used to resolve all queries related to distributing a capsules keys.

5.2 Capsule Generation and Key Distribution

The capsule generation process is outlined in figure 5.3. To create the capsule, the data owner transmits to the remote trusted server the data file, the policy (written in the policy API), and a list containing email addresses of people approved to receive the decryption keys. On receiving the request, the remote server creates an 128 bit AES key and a randomly generated UUID. This UUID is added to the capsule header.The header, the data file and the policy are merged and encrypted

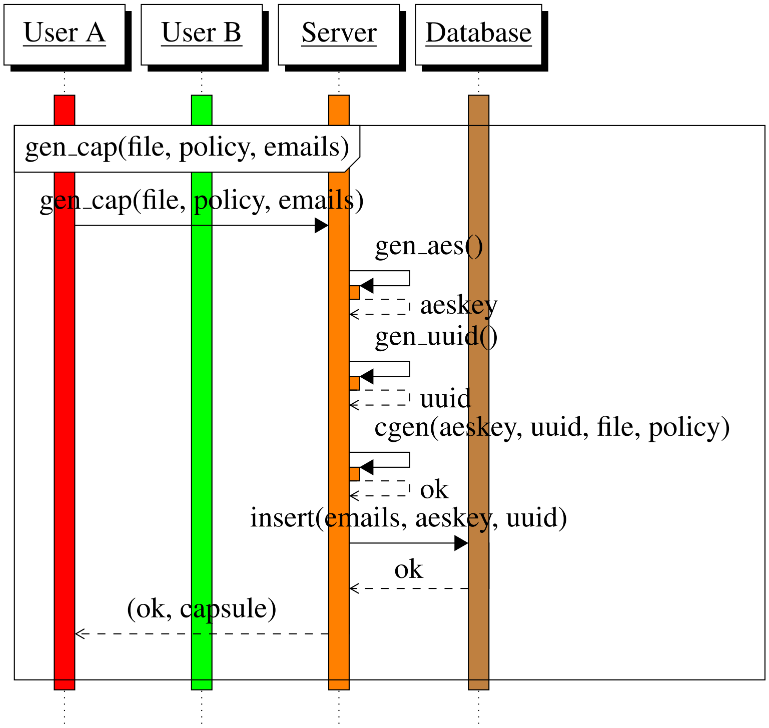
User A
User B
Server
Database
gen_cap(file, policy, emails)
gen_cap(file, policy, emails)
gen_aes()
aeskey
gen_uuid()
uuid
cgen(aeskey, uuid, file, policy)
ok
insert(emails, aeskey, uuid)
ok
(ok, capsule)

Figure 5.3: Capsule Generation and Key Registration

using the AES key that was generated.

Once the encryption step is complete, the <`UUID, AES key, approved email list`> tuple is persisted to the database. At the end of this process, the capsule thus created is returned to the user.

Once the capsule creation step is complete, the data creator can send the capsule to the designated recipients. When a capsule recipient wishes to open the capsule, a decryption process is triggered as illustrated in figure 5.4. The open call to the capsule file is mediated through the FUSE filesystem, which initiates a `capsule_open` call to the secure world. On receiving the `capsule_open` request, the Trusted Application searches secure storage for the AES key corresponding to the UUID found in the capsule header.

If no AES key is found, the secure world initiates a lookup request to the remote server by sending a `get_key` request with the caspule's UUID and the secure worlds RSA public key. The remote server looks at the registered capsules and

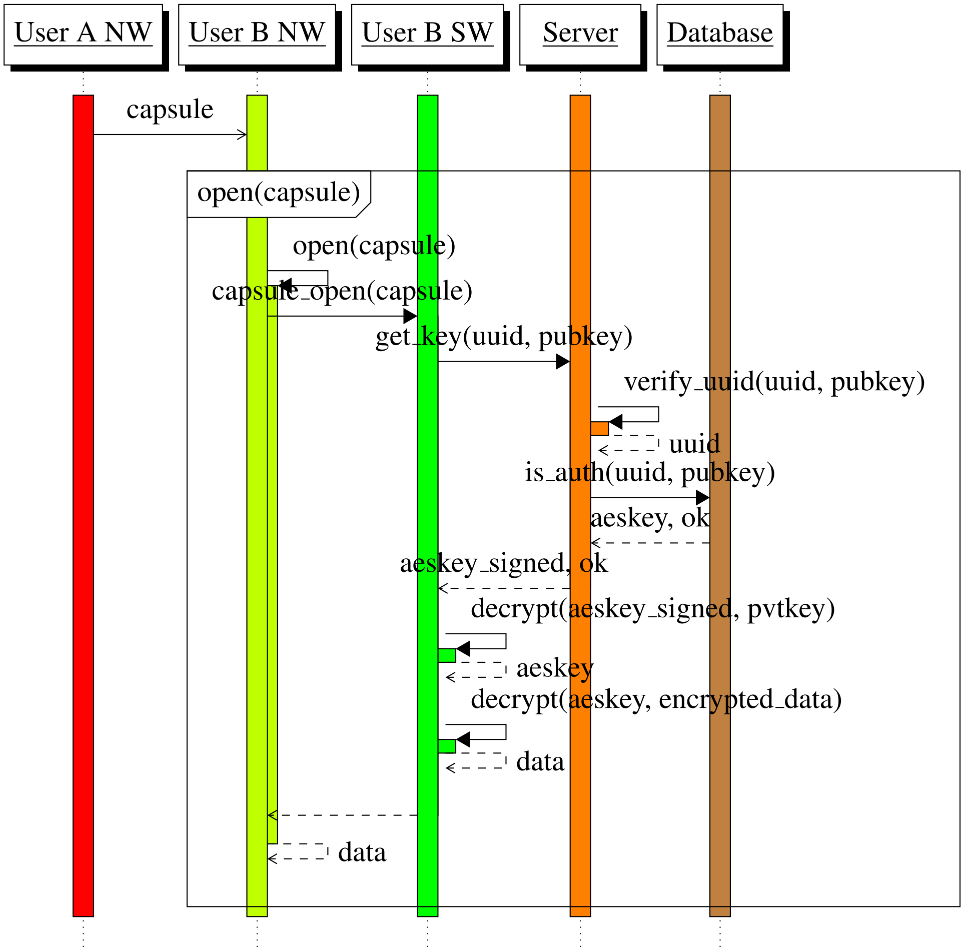

Figure 5.4: Capsule Decrypt as Recipient

verifies that the RSA public key belongs to an approved user. If it does, it returns the AES key encrypted with the RSA public key received by the server in the `get_key` request. On receiving the encrypted AES key from the remote server, the secure world uses it's private RSA key to decrypt the capsules AES key. This AES key is used to decrypt the capsule and return the data to the normal world.

Chapter 6

Use case examples

In this section, we discuss several use cases to highlight the capabilities of Trusted Capsules.

Access control based on time or location: Enterprises may wish to restrict employees from opening company files outside the office or a user may require his family members to view shared pictures only at their homes. Alternatively, the data owner may wish to allow access to sensitive content only within a predetermined time period. Such requirements are straightforward to express in our system. When a capsule policy's `evaluate_policy()` function is evaluated at the time of `open()`, it can access the device location and time [1] to decide if the access should be allowed or denied. Alternatively, instead of simply denying access to a capsule, policies may use the `redact()` API in Table 4.1 to allow access but with sensitive content redacted.

For example, Figure 6.1 illustrates a policy that denies access to the capsule if the location from which it is being accessed is outside the specified location range.

Requiring permissions in real time: In some cases, users may wish to have real-time control over their data. For example, Alvin may wish to be asked each time Barbara opens his capsule whether or not to allow her access. It is straightforward to support this scenario in Trusted Capsules as policies can communicate

[1] Information such as device location can only be trusted if the device driver is located within TrustZone and the peripheral bus allows exclusive access to the sensors. There has been prior work that achieves this. The Trusted Capsules prototype does not do this, and relies on the Normal World OS to provide this information. We leave this as future work.

with remote servers over the Internet.

We implemented this scenario in our prototype using Twitter. When a user opens a capsule, the policy uses the `getState()` API method to communicate with a custom server and passes the Twitter handle of the capsule owner. The server then sends a direct Twitter message to the owner of the capsule with an access request and asks him to respond with a “yes” or “no” to approve or decline the access, respectively. The server returns the owner’s decision to the policy and the appropriate action is taken. At the moment, the Twitter message to the owner does not identify the user trying to open the capsule but this can be implemented by mapping unique device identifiers to Twitter handles.

Progressive trust: The APIs in Table 4.1 may be composed to support other useful scenarios. Suppose Bob wants to share sensitive data with someone but does not yet completely trust that person. He can use a policy that contacts a remote server to log access attempts and to identify what data should be returned to the app. Initially, Bob may choose to provide a heavily redacted version of the data (e.g., an image with blurred-out faces or a document with key sections removed). As his trust towards the person grows, he can progressively share more sensitive content by recording his decisions on the server.

As an example of a policy with progressive trust, considers Figure 6.3 which consider content pre-distribution: a capsule creator writes this policy to pre-distribute their content while ensuring that the content cannot be viewed until a pre-set release date. For this use case, we rely on a trusted remote server for getting the time. Capsule metadata is first inspected using `getState()` to check if the content has already been approved for access by the policy. If this is indeed the first access to the capsule, using the `getTime()` API, the remote server is contacted to get the epoch value and it is compared to the epoch value in the policy. If the remote epoch stamp is greater than the time encoded in the policy, the access is approved, and the metadata is updated using `setState()` to reflect this. Any subsequent accesses to the capsule do not involve querying the remote server for getting the time.

Location based redaction: The policy as specified is fairly restrictive and blocks all access to the data contents outside a geographical range. There can be a scenario where the data owners wish to allow the partial revelation of the data in

```
longitude = 1250
latitude = 200
range = 10

function evaluate_policy( op )
    if op == POLICY_OP_OPEN or op == POLICY_OP_CLOSE then
        long, lat, err = getLocation( POLICY_LOCAL_DEVICE )
            if err ~= POLICY_NIL then
                comment = ”Failed to getLocation”
                return false
            end
            if math.abs(long - longitude) <= range
            and math.abs(lat - latitude) <= range then
                comment = ”GPS coordinates in range”
                return true
            else
                comment = ”GPS coordinates are not in range”
                return false
            end
    end
end
```

Figure 6.1: Simple location based access policy

locations that are not trusted. To fulfill this requirement, the policy in Figure 6.1 can be tweaked to allow redaction of data that is marked confidential by the creator.

The data creator can encapsulate sensitive information in the data within some policy defined secrecy tags (for example: `<secret> , </secret>` can be used). The `redact()` API can be used in the policy to look for these tags and replace the text contained therein with a replacement string. This policy is shown in Figure 6.2.

```
replaceVar = "REDACTED"
longitude = 1250
latitude = 200
range = 10
startTag = "<secret>"
endTag = "</secret>"
function evaluate_policy( op )
    if op == POLICY_OP_OPEN or op == POLICY_OP_CLOSE then
        long, lat, err = getLocation( POLICY_LOCAL_DEVICE )
        if err ~= POLICY_NIL then
            policy_result = POLICY_NOT_ALLOW
            comment = "Failed to getLocation"
            return
        end
        if math.abs(long - longitude) <= range
        and math.abs(lat - latitude) <= range then
            comment = "GPS coordinates in range"
            policy_result = POLICY_ALLOW
            return
        else
            comment = "GPS coordinates are not in range. Redacting data."
            policy_result = POLICY_NOT_ALLOW
        end
        while s ~= nil and e ~= nil do
            s = string.find(data, startTag, s)
            if s ~= nil then
                e = string.find(data, endTag, s+1)
                if e ~= nil then
                    err = redact( s, e, "replaceVar" )
                    if err ~= POLICY_NIL then
                        policy_result = err
                        return
                    end
                end
            end
        end
```

Figure 6.2: Location based redaction policy

```
-- remote server information
remote_server = "198.162.52.127:3490"
-- return keywords
policy_result = POLICY_NOT_ALLOW
comment = ""

-- policy-specific keywords
open_time = 1523338041
opened = "opened"

function evaluate_policy( op )
    if op == POLICY_OP_OPEN then
        r, err = getState( opened, POLICY_CAPSULE_META )
        if r == "true" then
            return true
        else
            curr_time, err = getTime( POLICY_REMOTE_SERVER )
        end
        if err ~= POLICY_NIL then
            policy_result = err
            comment = "Failed to get time from remote server"
            return false
        end
        if curr_time > open_time then
            err = setState( opened, "true", POLICY_CAPSULE_META )
            if err ~= POLICY_NIL then
                policy_result = err
                comment = "Failed to update capsule metadata"
                return false
            end
            return true
        end
    end
end
```

Figure 6.3: Policy to allow content pre-distribution

Chapter 7

Prototype

We prototyped Trusted Capsules on a LeMaker HiKey development board [6]. It has an octa-core ARM Cortex-A53 processor, 2 GB of RAM, 8 GB of flash storage. and it comes with TrustZone unlocked, thereby allowing us to control what OS runs on the TEE. We use Linaro OP-TEE OS (version 3.3) in TrustZone and a HiKey Debian OS (based on Linux 4.4.15) in the normal world. We modified the OP-TEE OS to implement several missing `libc` functions (such as `atoi` and `strcmp`). As the HiKey board does not have a GPS receiver, we mocked a GPS device that returns predefined longitude and latitude values.

Capsules are encrypted with 128-bit AES. We consider the distribution of keys required to decrypt capsules outside the scope of this paper.

Our data monitor is written in C and consists of about 6.2K SLOC: the policy execution engine, which runs within the TEE, has about 4.2K SLOC while the normal world framework has 2K.

7.0.1 Prototype Evolution

The system design and the prototype evaluated in this paper has evolved from a previous design of the system. This prior system ("version-0") had the ambitious goal of evaluating a Lua-based policy in TEE on all intercepted file I/O system calls on a capsule file: `open`, `close`, `read`, `write`, and `lseek`. As well, Version-0 revealed *chunks* of the file to normal world applications, rather than decrypting and revealing the entire file contents on `open`. Version-0 was not based on FUSE, but it used a custom system call interceptor in the normal world OS. This interceptor worked in a manner similar to the FUSE filesystem in our current design

Version-0 prototype was mature and stable, but had to be abandoned because of unacceptable application slowdown. This was due to the invasive nature of the system call handler that slowed down the behaviour of most applications that open and close many files at start-up.

More concretely, the time to open a small document under a no-op policy with FUSE on our hardware is 24ms, while the latency in Version-0 was 1.2s. This is a speed-up of 50x over Version-0.

The latency and throughput gap dramatically increased for large and complex file types, such as PDF JPEG. This can be observed in the raw video footage for several use-cases in Version-0 of the system: https://goo.gl/SiBEJB.

We note that while overhead in Version-0 was significantly better at the application layer as compared to the system call layer, nevertheless, the cost was prohibitive and was tightly connected to the policy being used. 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 extreme 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.

Our experiences with Version-0 of the Trusted Capsules prototype have been our guiding principle in making our current system perform better. Our benchmarking results (presented in the next Section) indicate which the current Trusted Capsules design, that evaluates policy exclusively on `open` and `close` calls, strikes a better trade-off between security and performance.

Chapter 8

Evaluation

We evaluated four aspects of our system: **(1)** the utility and simplicity of the policy language, **(2)** latency at the system call layer, and **(3)** the overhead associated with different policies. All performance evaluations were performed on our HiKey development board.

8.1 Policy language

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

We answered our first question by writing trusted capsule policies for the example use-cases from Section 6. For our second question, we measured the LOC for each policy that we wrote and show the result in Table 8.1.

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 time and location based policies[1] for usecases such as redaction and revocation.

[1] Refer Chapter 9 for a discussion about extending the Policy API.

Policy	LOC
Location Based Access (Fig 6.1)	30
Location Based Redaction (Fig 6.2)	36
Content Distribution (Fig 6.3)	28

Table 8.1: LOC for example policies from Section 6.

8.2 System call microbenchmarks

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

Are operations on regular files affected? We measured the latency of filesystem operations for a regular file and a capsule. Since our system is based on FUSE, we evaluate the performance of the Trusted Capsule system by comparing against system call latencies for a regular file on the same mountpoint.

We found that the performance of system calls on regular data is only affected on *open* syscall. This is due to the overhead of checking whether the target file is a trusted capsule.

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 1000 runs. For throughput measurements, we randomly read and wrote 4KB of data to a trusted capsule for 10 seconds. To get an estimate of performance on the first use, we repeat the experiment with a cold cache achieved by dropping the page cache. For each test trusted capsule, we attached an empty null policy that always evaluated to true. We present our results in Figure 8.1 and 8.2.

The latency for `open` and `close` operations for a capsule present a prominent spike when compared to the operations on regular files. This is expected since our current prototype interposes on only these operations. An `open` operation on a null-policy capsule (warm cache) completes in 23 milliseconds compared to the 11.7 milliseconds for a regular file. The close operation on a capsule completes in

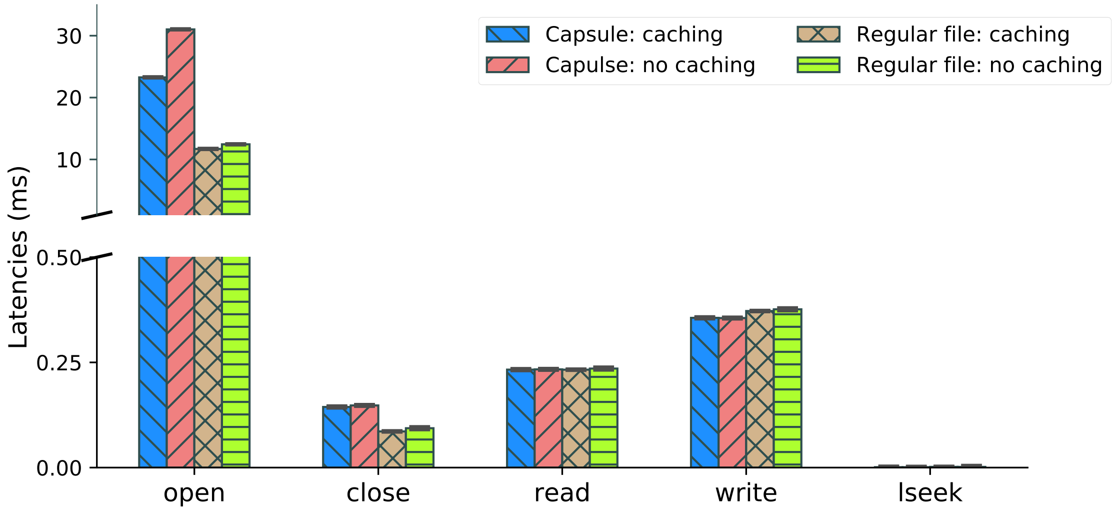

Figure 8.1: Average system call latency

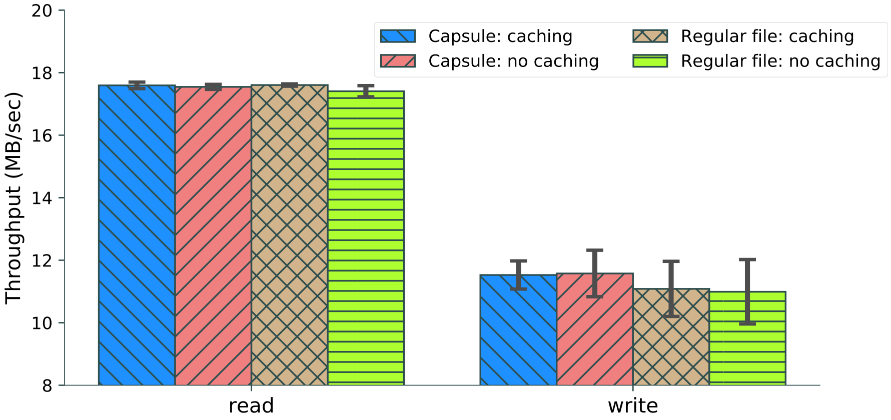

Figure 8.2: Throughput of Read and Write operations to a capsule

144 microseconds as compared to 86 microseconds for a regular file.

The observed latency spike is more pronounced for `open` than for `close`. We understand this to be a direct result of the greater number of steps that have to happen in TrustZone to initialize the Trusted Application, which do not need to be done while servicing `close` call on a capsule.

We were able to achieve 17.59MB/s throughput for reading and 11.52MB/s throughput for writing to a no-op capsule on a warm cache. This is comparable to the read (17.6 MB/s) and write throughput (11.1 MB/s) achieved for a regular file when accessed in the same experimental setup. When the same experiments were repeated for a cold cache, the throughput drops marginally.

The read and write throughputs for a capsule, as compared to a normal file, were expected to be nearly identical. This is expected in our system since all reads and writes to a capsule gets directed to a shadow file, which is treated like a regular file in FUSE.

8.3 Policy Performance Evaluation

In this section we present our findings on the impact that policies of varying complexity have on the performance of the system. To measure the overhead associated with the policy execution, we compare the latency microbenchmarks for `open` operations for a policy containing capsule, normalized with respect to the latency for opening a null-policy capsule. These results are presented in Figure 8.3. There is a sharp increase in the latency when there is a non-null policy being evaluated, and this latency increases with the complexity of the policy.

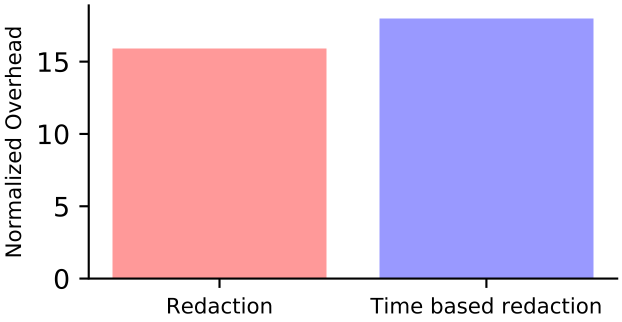

Figure 8.3: Normalized latency of servicing an `open` for different policies with respect to the latency to service a null-policy capsule open request.

Figure 8.3 compares two policies: a redaction policy that redacts sensitive tags without performing other checks and a local time based redaction policy, which performs redaction based on the epoch value obtained from the device. The redaction policy uses the `redact()` API from Table 4.1, while the Time based redact policy uses the `redact()` API as well as well as the `getTime()` API. This extra work to service an open request is evident from Figure 8.3.

We believe that such performance degradation can be mitigated with more efficient policy code, for example policies that run at coarser granularity or use caching to mitigate expensive checks.

Chapter 9

Limitations

Now we turn to discuss the design limitations of Trusted Capsules. In this section, we cover the limitations imposed by our design choices and the limitations imposed by the specific choice of software and hardware. Then later on, in Chapter 10, we take a gander at why Trusted Capsules remains a rather naive attempt at solving the problem of retrofitting existing applications with security extensions.

9.1 Design Limitations

1. **Inability to limit trust in optimistic state**: In the optimistic state, we trust the normal world kernel, the app, and the user, to not leak capsule data to unauthorized apps. Such trust may not be warranted even in a non-adversarial setting. For example, an app might create temporary copies of the files it has opened into a world-readable directory or the user might copy the data into the system clipboard. While we may use techniques such as information flow control to detect such data leaks, doing so would dramatically reduce performance.

2. **Lack of app semantics**: Since we interpose only on the `open()` and `close()` syscalls to execute policies, a policy may not reason about *why* an app is opening a file. For example, when a user opens a document in a `vim`, `vim` opens the document, creates a swap file, copies the contents of the document to the swap file, and then closes the document. All subsequent reads

and writes are done to the swap file thereafter. On close, the original file is opened again, the contents of the swap file are copied back to the main document, and the main file is closed and the swap file deleted.

Hence, while from a user's perspective, the capsule was opened once and closed once, the policy would observe multiple capsule access attempts. This makes it difficult to identify the legitimate accesses from the application quirks. The data monitor handles Policies that rely on access logs have to be aware of this disconnect.

3. **Abusive policies**: Although we run capsule policies in a sandbox, we do not completely prevent all damage a malicious policy can inflict. It can, for example, access a user's GPS data and send them to a server to track her whereabouts. To handle this limitation, we either need some systematic way of vetting the data a policy sends to a remote server or prevent it from sending device data altogether.

4. **Trusting actions from untrusted OS**: In our design, the signal (`open()` and `close()`) to the TEE to decrypt a capsule originates in the untrusted part of the system. The TEE cannot differentiate between genuine access and similar access requested by a malicious application.

   Trusting the actions originating in the normal world dilutes the guarantees we make about the transition from the pessimistic state to the optimistic state of our threat model. Moreover, trusting the normal world invalidates the need to use a TEE for secure processing of a Trusted Capsule.

9.2 Prototype Limitations:

In this section, we list the limitations that bound the current prototype from realizing the full vision of the Trusted Capsule model of data protection. Here we note some design and implementation limitations.

1. FUSE can be used interpose on only the file I/O system calls that are directed to a FUSE serviced mount point. This poses some challenges in making Trusted Capsules work seamlessly with an unmodified application.

There is no way for the FUSE filesystem code to identify when a process that had been issuing IO to the mount point dies. The implication of this fact for the prototype is that there is no good and atomic way to delete the shadow file on the termination of the process. The prototype handles this by setting up a background process that monitors if the process that had accessed the mount point has terminated.

2. The stock configuration for the TrustZone memory partitions makes the Secure World a very memory-constrained environment. The memory that is available in the secure world is only 10 MB, and that needs to host the Secure OS as well as any trusted application code that must run in Trustzone. Linaro OP-TEE recently included dynamic shared memory in their Secure OS, but to access those features, one has to have a higher kernel version than what gets shipped with the stock Debian OS rootfs image. More recent Linux kernel versions have known problems with the HDMI drivers, which causes the Linux kernel to panic when a monitor is plugged in.

   This limitation hits our prototype particularly badly. In the current prototype, we send a copy of the entire file to TrustZone to decrypt. Since there are severe restrictions on the amount of memory available in TrustZone, we are unwittingly bounded on the maximum file size that can be processed as a capsule.

3. The implementation has a limitation that it needs to create copies of the data buffers to process each part of the capsule. This creates more memory pressure in an already resource-constrained environment.

4. The Policy API is currently limited to policies that require access to local time and the GPS coordinates (ideally, the GPS driver should run in the TEE OS). To support policies that rely on other signals and/or sensors will require adding the device driver, writing the interfacing code in the trusted application, and updating the Lua-to-C bindings. This is a well-defined albeit convoluted workflow that makes it harder to make the policy language more flexible.

Chapter 10

Securing Applications

TrustZone was first announced in the year 2004 [13], and has since been used for a diverse range of applications as can be seen in section 11. Some of these applications target new use-cases that can benefit from the guarantees a TEE provides, while in other cases, pre-existing applications are re-structured and re-factored to make use of a TEE. How these projects structure their secure applications differ based on the information being protected and the threat model around which the application has been designed. We first investigate the patterns in prior work, and then analyse why these are not the right fit for Trusted Capsules.

10.1 Modifying Applications to Use a TEE

We begin by inspecting how prior research in the space of TEE-enabled security have structured their applications. We have identified four distinct patterns that prior research has taken in trying to structure their secure applications, and these evolve based on the threat model that the application is being designed to protect against.

1. **Manually Split the Application into Trusted and Untrusted Parts:** This is the most commonly used methodology designing applications with the view to use a TEE to safeguard some aspect of the program state [14, 46, 59, 62].

   The basic idea is that there is some easily identifiable functionality that can

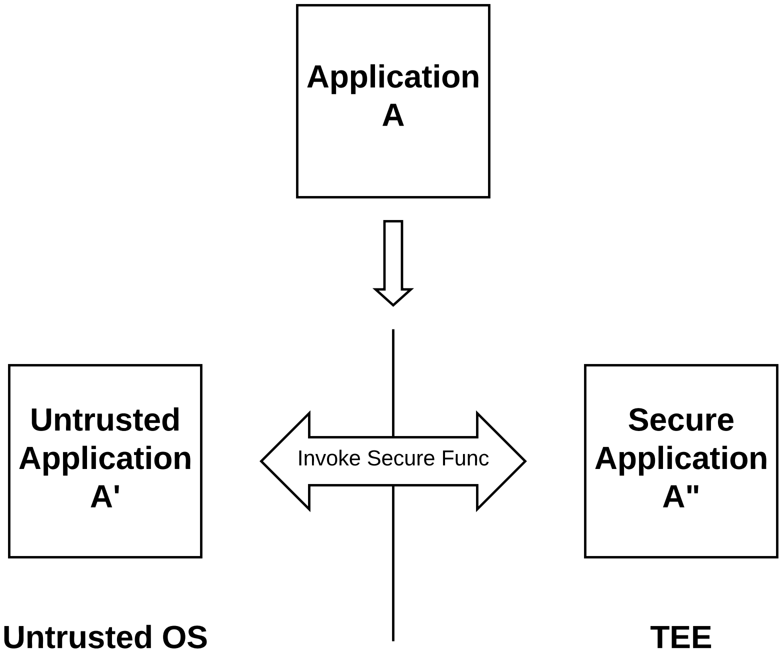

Figure 10.1: An application can be split into two parts - one that resides in the untrusted operating system and has the interfaces to secure functionality that resides in the trusted environment.

be plucked out of the monolithic application and can be offloaded to the TEE. The rest of the application that remains on the untrusted OS only has interfaces to the secure functions in the TEE. This is shown in Figure 10.1. The TEE in this scenario could either be a trusted operating system that is hosted as a virtual machine or it could be a more conventional hardware-based TEE like Intel's SGX or ARM's TrustZone.

This application splitting paradigm is promising in scenarios where the critical functionality that needs protection is small, well defined, and can be extracted out into a TEE-resident service. Prime examples of this are biometric verification, cryptographic functions, payment processing, and verifying the user's action and intent. An implicit assumption with this strategy is that the partitioning of the application is a deliberate decision that is made at the time of designing the application and therefore can not be used for an unmodified

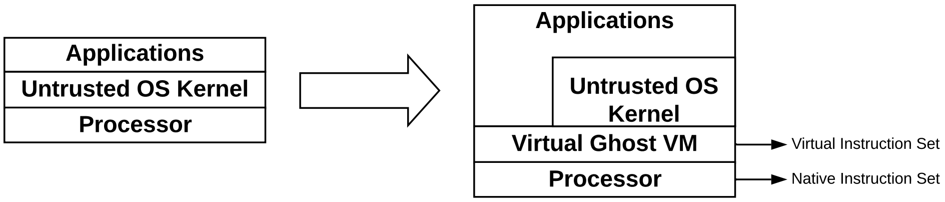

Figure 10.2: Virtual Ghost uses LLVM to create an intermediate layer the OS and the processor to protect the application from the unfettered access otherwise enjoyed by a kernel

application.

2. **Compiler Supported Application Sequestering**: There has been work in securing application using compiler wizardry. VirtualGhost [19] defines a compiler-based instrumentation of the kernel and the application to protect the applications data confidentiality and integrity. The OS is compiled to a virtual instruction set which is handled by the Virtual Ghost VM as shown in Figure 10.2. This "virtual machine" is used to limit the accesses the kernel has into the state of the application. This is an interesting work that provides strong guarantees about the application's integrity and data confidentiality without using a hardware-based TEE.

   There has been work in automatically identifying which parts of the application need to be protected and splitting the application using the TEE API [44, 56]. This approach involves using static analysis and dataflow analysis along with optional annotations in the program to automatically identify the partitioning scheme, and then refactoring the application into two parts that are bridged using the TEE API. Programming language theory can be used to demonstrate equivalence between the split halves and the original application.

   These approaches might be an efficient way to retroactively make an unmodified application ready for a TEE.

3. **Complete Isolation of the Application Inside the Secure World** This approach is, in theory, the path of least resistance to use a TEE. There have been

some projects that try to contain an entire application in the TEE - Graphene [63, 64] and TrustShadow [27], for example. These systems rely on the TEE to make available all services that the application might require - most importantly the I/O and peripheral access and the system call interface. This approach, while promising, is fraught with its own set of problems. TEE's are a restrictive execution environments - Intel SGX, for example, does not provide access to a system call interface. No application running in SGX is allowed to make a system call. Similarly, in TrustZone and specifically OP-TEE, because the TEE OS is minimalist, it is difficult to support a full-fledged application.

Graphene [64] provides good insight into the TEE software design space. While designing any secure application, the developer has to decide between - pulling too much functionality into the TEE vs keeping the Trusted Compute Base minimal and simplifying the application to require lesser of the runtime vs making a rich run time; application changes to make it more amenable to the TEE runtime. These are challenging choices to make and rely not on the application but the baseline runtime he/she is working with.

4. **System Call Interception** If the complete isolation of an application is not possible, and it is necessary to maintain some part of the application in the untrusted OS, the most enticing option available to the security engineer is to use system call interception in the normal world OS and handle the "sensitive" accesses to data and the network in the TEE. This approach can go awry if there is not a strict control on the number and type of system calls that are handled in the TEE - the more elaborate the handler in the TEE, the higher will be the slowdown, and less usable will be the system. TrustedCapsule's first prototype was overly ambitious in the amount of work that was being done on every single system call to a capsule file, causing in some cases a 300x slowdown on simple tasks such as opening a capsule.

10.2 Securing Unmodifed Applications

While the TEE OS can be extended to include all the services that a rich application would provide, it defies the rationale behind having a small trusted computing base. This limits the range of applications that are a good fit for running in the TEE. Applications that are computation heavy and need little or no I/O are good fits for this - Machine Learning tasks [52] and Cryptography and key management[4, 12].

For applications that *have* to run I/O to be useful - which is a lot of user-centric applications such as word-processing systems, image viewers and other such interactive applications. These interactive applications have their utility in revealing information to the user and allowing the user interact with the data on demand. When the user interaction is an indispensable part of the application and its utility, none of the application structuring schemes lead to completely satisfying outcomes. This problem is exacerbated when the TEE usage is provided as an overlay scheme atop an existing application to provide backward compatibility and ease of deployment[1].

These secure overlay schemes do not align with any application structuring methodologies that researchers have come up with. Supporting a full-fledged application is difficult because there are a lot of quirks that need to addressed and at the same time, the technique needs to be generic enough to support a wide range of applications. The goal of keeping the interface generic and the application unmodified limits the security guarantees that can be made. For example, in the case of TrustedCapsules, because the application still receives all the decrypted data back from TrustZone, there can be no further guarantees made after the first file access from the normal world. Moreover, the policy engine in the secure world needs to reason about the state of the normal world, which is problematic because such information is gleaned from the normal world. A malicious normal world user could fake these state values (for example, the GPS coordinates) while requesting the decryption, and because the policy engine relies on the veracity of these signals, it could be tricked into incorrect policy evaluation.

[1] The FUSE-based interception scheme is an overlay on top of regular file access that unmodified applications perform. Using such an overlay scheme has an advantage that it doesn't require a special file viewer for a capsule file and provides a better user experience.

If we place trust in the normal world, we don't need a TEE. If we don't implicitly trust the normal world, then the data should never be revealed back to the normal world. This cyclical conundrum can only be resolved by refactoring the application in a way that uses the TEE by following the design patterns mentioned earlier in section 10.1.

Chapter 11

Related Work

Securing data with policies: The concept of associating policies to data with authenticate accesses to that data is not new. An early expression of this is XACL, which specifies access control policies within XML documents [28]. Karjoth et al. proposed using *sticky policies* to provide enterprises better oversight over the customer data they collect [34]. These policies capture customer-specified requirements (e.g.: “delete my data after 30 days”) and are associated with the collected data. They are then enforced cooperatively within the enterprise as the data is used. Subsequent work strengthened this scheme by encrypting the data bundled with the policy using IBE (identifier-based encryption) and decrypting it only if its policies are satisfied [50, 53]. Encrypting the data reduces the need for cooperation and allows sharing data across enterprise boundaries

Maniatis et al. outlined a vision that allows *all* users to protect their data before they share them across machine boundaries [48]. Their conceptual architecture uses the sticky policy approach to package data in units known as *data capsules*. When an application needs to use a capsule and satisfies the capsule’s policies, an abstract secure execution environment decrypts the capsule and executes the application. An implementation of this architecture was left as an open question.

More recent works use trusted computing features on mobile devices to protect data with the sticky policy approach. Li et al. proposed DroidVault to allow employees in an enterprise to securely store and process sensitive company data on their untrusted Android devices [43]. Its architecture only allows trusted code signed by the enterprise to operate on the data and executes it in ARM TrustZone. To display data and receive user inputs, it relies on secure I/O between the peripherals (display, keypad, etc.) and TrustZone. This architecture ensures unencrypted versions of the sensitive data do not leave the TrustZone environment. Lazouski et al. proposed using TPMs (Trusted Platform Modules) to ensure only vetted versions of the OS and applications are loaded before accessing sensitive data and executing their policies [39]. In principle, this approach allows policy execution and data access in the normal world (outside TrustZone) while guaranteeing the absence of malicious applications.

Other related works in this area include Excalibur, which enables a cloud provider to protect data stored in its cloud from being exfiltrated by its administrators who have access to the cloud management interface [58]; PCD (policy-carrying data), which lets an end-user attach terms of service to his data before sharing it cloud service providers and thereby disincentivizing them from misusing the data [60]; Ryoan, which enables users to submit their sensitive data to a cloud service provider for processing without requiring either the user to disclose the data or for the provider to release their proprietary code [31]; and P3, a private photo-sharing service that protects images shared by users from untrusted service providers [54].

Trusted Capsules differs from these in its aim and scope: it uses the sticky policy technique to allow end-users to protect their data as they share it with other end users and unlike P3, it is data type agnostic. While Trusted Capsules uses ARM TrustZone to securely execute the policies, it allows unvetted normal world processes to access unencrypted sensitive data in the optimistic state (unlike DroidVault and the work by Lazouski et al.). Our approach is motivated by usability concerns as we want authorized users to be able to use their desired third-party apps to process sensitive data.

There are now startups that have emerged as players in the domain of providing data security systems. A startup called Sandstorm [9] 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.

Information Flow Control based mechanisms: There has also been a vast body of research that studies providing data confidentiality through label-based solutions such as Distributed Information Flow Control [18, 21, 38, 51, 55, 66, 67]. 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.

In DIFC, labels create a natural ecosystem for composition that allows 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. On the other hand, trusted capsules are backward compatible and do not require constructing a complex security lattice as in DIFC.

Another popular approach is tainting [23, 24, 30, 68]. It tracks information flow by interposing on the system operations at the instruction-level. These solutions can track the flow of information at an extremely fine granularity. resource-intensive, both in memory and CPU.

Policy Based Isolation Mechanisms: Traditional isolation-based solutions remain one of the most widely used practical solutions currently to provide data protection. These solutions, such as VPN, VMWare Ace [1], Secure Spaces [10] and Hypori [7], attempt to prevent sensitive data from leaving in the first place by enforcing policy at the network boundary between external and internal systems. In these cases, policies that restrict the movement of sensitive data can still be defeated by transformations, such as encryption and compression. Also, 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 [29, 40], limiting data lifetimes [20, 33] and providing recourse actions such as backtracing intrusions [25, 35].

Other TEE work: The research community has used TEEs such as ARM TrustZone and Intel SGX for a variety of purposes - to provide a secure environment for running VMs, secure partitions or executing parts of third-party applications and to store their data [22, 37, 59], to provide a root-of-trust for performing runtime measurements [15–17, 61] and to secure peripherals [47]. In general, these are orthogonal to Trusted Capsules.

VButton uses TrustZone to attest whether the UI inputs on the smartphone were initiated by the user [42]; SeCloak provides direct control (on/off) over device peripherals even when the normal world OS is compromised [41]; Truz-Droid enables users to securely input and send secrets e.g., login credentials, to authorized servers without executing third-party code in TrustZone [65]; TrustShadow protects applications from untrusted OSes by executing them with a runtime in TrustZone [26]; and SchrodinText allows the untrusted normal world OS to render sensitive text in the display received from an application backend server without revealing the contents of the text [57]; DelegaTEE, which uses Intel SGX to enable users to share their access to online service providers without revealing their credentials [49].

Chapter 12

Conclusion

Data security on remote devices that the data owner cannot control represents 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. → page 52

[2] Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware. Accessed: 2019-02-15. → page 8

[3] Cryptomator - free cloud encryption. =https://cryptomator.org/. → page 1

[4] Security solutions based on runtime encryption® platform. =https://fortanix.com/solutions/. → page 48

[5] Global platform api specifications. http://www.globalplatform.org/. Accessed: 2019-02-15. → page 9

[6] LeMaker HiKey. http://www.lemaker.org/product-hikey-index.html. → pages 6, 34

[7] Hypori. http://www.hypori.com/. Accessed: 2019-02-15. → page 52

[8] ARM mbed TLS. https://tls.mbed.org. → page 18

[9] Sandstorm. https://sandstorm.io/. Accessed: 2019-02-15. → page 51

[10] Secure spaces. https://www.spacesmobile.com/. Accessed: 2019-02-15. → page 52

[11] When does snapchat delete snaps and chats? =https://support.snapchat.com/en-GB/article/when-are-snaps-chats-deleted. → page 1

[12] Web cryptography api. =https://www.w3.org/TR/WebCryptoAPI/. → page 48

[13] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004. → pages 4, 44

[14] A. Amiri Sani. Schrodintext: Strong protection of sensitive textual content of mobile applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 197–210, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4928-4. doi:10.1145/3081333.3081346. URL http://doi.acm.org/10.1145/3081333.3081346. → page 44

[15] 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. → page 53

[16] 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.

[17] 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. → page 53

[18] 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. → page 52

[19] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *ACM SIGPLAN Notices*, volume 49, pages 81–96. ACM, 2014. → page 46

[20] 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. → page 52

[21] 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. → page 52

[22] 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. → page 53

[23] 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. → page 52

[24] 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. → page 52

[25] 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. → page 52

[26] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone. In *Proceedings of MobiSys '17*, June 2017. → page 53

[27] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. *arXiv preprint arXiv:1704.05600*, 2017. → page 47

[28] S. Hada and M. Kudo. XML Access Control Language: Provisional Authorization for XML Documents. http://xml.coverpages.org/xacl-spec200102.html. → page 50

[29] R. Herbster, S. DellaTorre, P. Druschel, and B. Bhattacharjee. Privacy capsules: Preventing information leaks by mobile apps. In *Proc. of MobiSys*, 2016. → page 52

[30] 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. → page 52

[31] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of OSDI '16*, November 2016. → page 51

[32] M. B. D. T. Joakim Bech, Ard Biesheuvel. Implications of meltdown and spectre : Part 2, Feb 2018. → page 6

[33] J. Kannan and B.-G. Chun. Making programs forget: Enforcing lifetime for sensitive data. In *HotOS*, 2011. → page 52

[34] G. Karjoth, M. Schunter, and M. Waidner. Platform for Enterprise Privacy Practices: Privacy-enabled Management of Customer Data. In *Proceedings of PET '02*, April 2002. → page 50

[35] S. T. King and P. M. Chen. Backtracking intrusions. *ACM SIGOPS Operating Systems Review*, 37(5):223–236, 2003. → page 52

[36] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019. → page 6

[37] 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. → page 53

[38] 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. → page 52

[39] A. Lazouski, F. Martinelli, P. Mori, and A. Saracino. Stateful Usage Control for Android Mobile Devices. In *Proceedings of STM '14*, September 2014. → page 51

[40] 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. → page 52

[41] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. SeCloak: ARM Trustzone-based Mobile Peripheral Control. In *Proceedings of MobiSys '18*, June 2018. → page 53

[42] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proceedings of MobiSys '18*, June 2018. → page 53

[43] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena. DroidVault: A Trusted Data Vault for Android Devices. In *Proceedings of ICECCS '14*, August 2014. → pages 1, 50

[44] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 285–298, 2017. → page 46

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018. → page 6

[46] D. Liu and L. P. Cox. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 7. ACM, 2014. → page 44

[47] 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. → page 53

[48] P. Maniatis, D. Akhawe, K. Fall, E. Shi, and D. Song. Do You Know Where Your Data Are? Secure Data Capsules for Deployable Data Protection. In *Proceedings of HotOS '11*, May 2011. → page 50

[49] S. Matetic, M. Schneider, A. Miller, A. Juels, and S. Capkun. DelegaTEE: Brokered Delegation Using Trusted Execution Environments. In *Proceedings of USENIX Security '18*, August 2018. → page 53

[50] M. C. Mont, S. Pearson, and P. Bramhall. Towards Accountable Management of Identity and Privacy: Sticky Policies and Enforceable Tracing Services. In *Proceedings of DEXA Workshop '03*, September 2003. → page 50

[51] 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. → page 52

[52] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 619–636, 2016. → page 48

[53] S. Pearson and M. C. Mont. Sticky Policies: An Approach for Managing Privacy across Multiple Parties. *IEEE Computer*, 44(9):60–68, 2011. doi:10.1109/MC.2011.225. URL https://doi.org/10.1109/MC.2011.225. → page 50

[54] M.-R. Ra, R. Govindan, and A. Ortega. P3: Toward privacy-preserving photo sharing. In *Proceedings of NSDI '13*, April 2013. → page 51

[55] 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. → page 52

[56] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 923–934. IEEE, 2016. → page 46

[57] A. A. Sani. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proceedings of MobiSys '17*, June 2017. → page 53

[58] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proceedings of USENIX Security '12*, August 2012. → page 51

[59] 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 44, 53

[60] S. Saroiu, A. Wolman, and S. Agarwal. Policy-Carrying Data: A Privacy Abstraction for Attaching Terms of Service to Mobile Data. In *Proceedings of HotMobile '15*, February 2015. → page 51

[61] 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. → page 53

[62] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 279–292, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298482. → page 44

[63] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi:10.1145/2592798.2592812. URL http://doi.acm.org/10.1145/2592798.2592812. → page 47

[64] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017. → page 47

[65] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proceedings of MobiSys '18*, June 2018. → page 53

[66] 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. → page 52

[67] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *NSDI*, volume 8, pages 293–308, 2008. → page 52

[68] 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. → page 52