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)

September 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. In this paper, 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

The lay or public summary explains the key goals and contributions of the research/scholarly work in terms that can be understood by the general public. It must not exceed 150 words in length.

Preface

This thesis is an original, unpublished work by Puneet Mehrotra, written under the supervision of Ivan Beschastnikh.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Thank those people who helped you.
Don’t forget your parents or loved ones.
You may wish to acknowledge your funding sources.

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 to encrypt their files before backing it up to the cloud or SnapChat to send self-destructing images that are viewable only for a limited period of time. 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 [36], 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

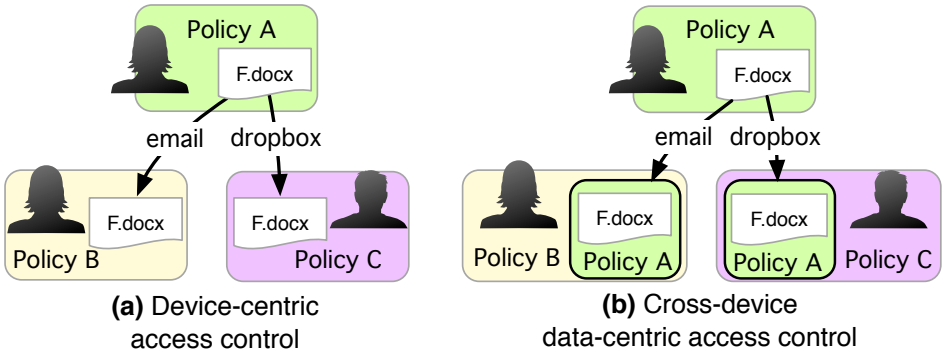

Figure 1.1: (a) Today, a data creator has no control over their data on remote devices: devices enforce local policies on data they receive. We propose (b) cross-platform policies that move with data and are enforced uniformally across devices.

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.

In this paper, we describe a platform-level file protection mechanism that does not restrict the apps users may use. To this end, 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 **??**.

We contribute **Trusted Capsules**, a data-centric access control abstraction that embodies the above 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 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 (1.96x and 1.67x, respectively, using a no-op 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 **TEE!** (**TEE!**) that is resistant to potential compromise of both applications and **OS!** running on the remote device. We use ARM TrustZone technology as our **TEE!** and Linaro OP-TEE as our **TEE!** low-level software stack. Within this **TEE!**, we handle sensitive cryptographic operations, perform policy evaluation, securely store policy state, and anchor a secure channel to the policy coordinator.

2.1 TrustZone

ARM TrustZone [9] is widely available on current commodity ARM processors. TrustZone physically partitions the CPU, memory and peripherals into two isolated logical "worlds" – normal and secure. Each world has its own banked system registers and MMU. To isolate the two worlds, all communications must pass through a small and heavily verified *secure monitor* gate. To facilitate a *world switch*, a special *smc* instruction is used to trap into the secure monitor. The secure monitor saves the banked registers (e.g., return address, stack pointer) of the calling world and loads the banked registers of the callee world prior to executing *eret* to return to the last execution point in the callee world.

Where the *smc* traps to is controlled by the secure world through its exception table register – VBAR, which holds the memory address of the exception table. The memory that holds the exception table can also only be accessed by the secure world.

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

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

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

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

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

The secure/normal paradigm operates orthogonally to the traditional concept of privilege levels, see Table 2.1. The secure monitor operates in secure mode at the highest privilege level (EL3), while untrusted application code and privileged normal world **OS!** operate in non-secure mode. The secure mode at privilege level EL0 and EL1 is reserved for trusted applications and the trusted **OS!**.

Architecturally, the privilege level of the CPU is controlled by a system register

	Secure	Normal
EL0	Trusted Application	Application
EL1	Secure **OS!** (**OS!**)	Normal **OS!**
EL3	Secure Monitor	-

Table 2.1: ARM processor modes.

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

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

2.2 Linaro OP-TEE

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

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

2.2.1 ARM Trusted Firmware

ARM Trusted Firmware (ATF) [2] is a set of reference boot and runtime firmware designs for ARM TrustZone. It initializes the secure world through a

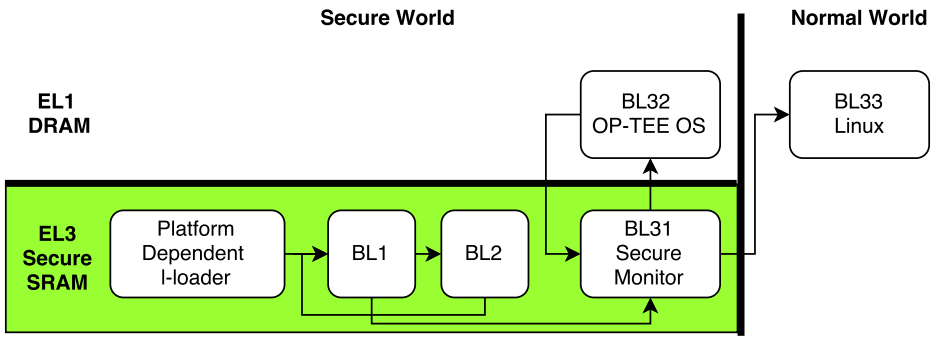

Figure 2.1: ARM TrustZone Boot Sequence.

multi-staged boot sequence, as shown in Figure 2.1. 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 capable of multi-threading, memory management, and running and isolating trusted applications. OP-TEE OS does not have a scheduler. It operates as a slave in a master-slave relationship with the normal world OS. Therefore, OP-TEE OS can only simultaneously run as many trusted application instances as there are cores at any given time. On multi-core architectures, each CPU can independently perform a world switch. When an interrupt occurs that needs to be handled by the normal world, OP-TEE OS transitions back into the normal world and once the interrupt has being handled, returns to its last point of execution within the secure world. Communication between the normal world and secure world occurs through a piece of pre-allocated shared memory accessible by both worlds. The shared memory is allocated by the secure world but is managed by the normal world. The secure world OS may access peripherals under the normal world's control and allocate shared memory through RPC calls into the normal world. For example, OP-TEE OS uses these RPC calls to access the normal world file system, with which it implements secure storage using a provisioned root key.

OP-TEE OS provides useful abstractions to build trusted applications that run in secure world user space (Secure EL0). Trusted applications can be single-instance or multi-session. OP-TEE OS applications conform to the GlobalPlat-

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

Table 2.2: Linaro OP-TEE API. Internal APIs are used by trusted applications and are prefixed by *TA_*. Client APIs are used by the normal world and are prefixed by *TEEC_*.

form Internal API [3] where each trusted application must implement a set of well-defined functions as entry-points. Client applications in the normal world invoke these functions through a similar set of GlobalPlatform Client API [3]. The list of functions are listed in Table 2.2. We use these APIs and secure storage provided by OP-TEE OS to build our multi-session trusted capsule application at the core of our trusted capsule monitor. Any call into trusted applications from the normal world are serialized on the normal world side by the TrustZone device driver.

2.2.3 OP-TEE Linux Driver

OP-TEE Linux Driver provides the normal world **OS!** (Linux) access to TrustZone. It represents TrustZone as a device file, which can be accessed from the normal world through the set of APIs listed in Table 2.2 from both user and kernel space. The TrustZone driver is responsible for two main tasks – (1) calling

into trusted applications running in TrustZone and (2) handling RPC requests from OP-TEE OS (e.g., file system, shared memory allocations). For trusted capsules, we extended the limited set of RPC calls available to the OP-TEE OS to include networking and direct file system operations. The TrustZone driver executes RPCs by using the OP-TEE Supplicant.

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

2.2.4 OP-TEE Supplicant

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

Chapter 3

Threat Model

3.1 Contextual Theat Model

Trusted Capsules use a contextual threat model. 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 on her device, 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; and 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-

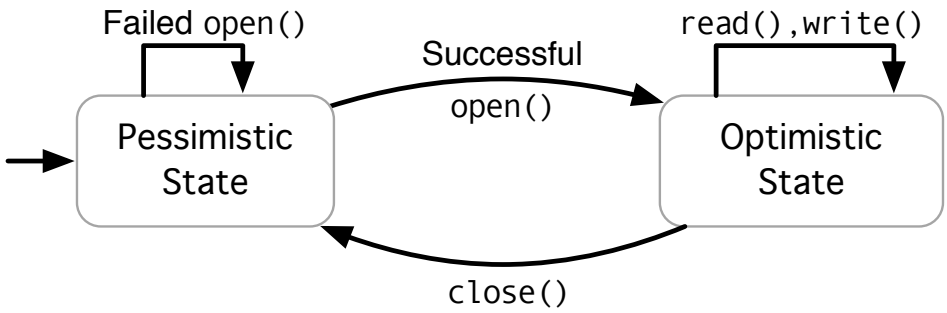

Figure 3.1: A finite-state machine view of Trusted Capsule's threat model. Each capsule on a device begins in the pessimistic state. A successful transition from the pessimistic to optimistic state means an app on the device tried to open the capsule and the capsule's policy authorized the access. Only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

We consider side-channel and analog attacks out-of-scope.

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 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

Adversary \ State	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.

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's aim is 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

At a high-level, Trusted Capsules packages data into protected units known as *capsules*. When an app in the normal world tries to open a capsule, the Trusted Capsules data monitor intercepts the open and executes the policy within the TrustZone TEE. If the capsule's policy authorizes the access, the capsule data is decrypted and returned to the app. When the application eventually closes the file, the data monitor re-seals the capsule, evaluating another on-close policy to determine e.g., whether capsule content can be mutated. In the rest of this section, we describe the components of our system.

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 followed by an encrypted data block. The header identifies the file as a capsule and contains integrity metadata used by the data monitor, such as a hash of the contents of the data block before they were encrypted. The data block contains the protected data, its access policy, and metadata associated with the policy, such as access logs. We assume that the cryptographic keys required to decrypt capsules are securely loaded into a secure storage area accessible only by the TEE.

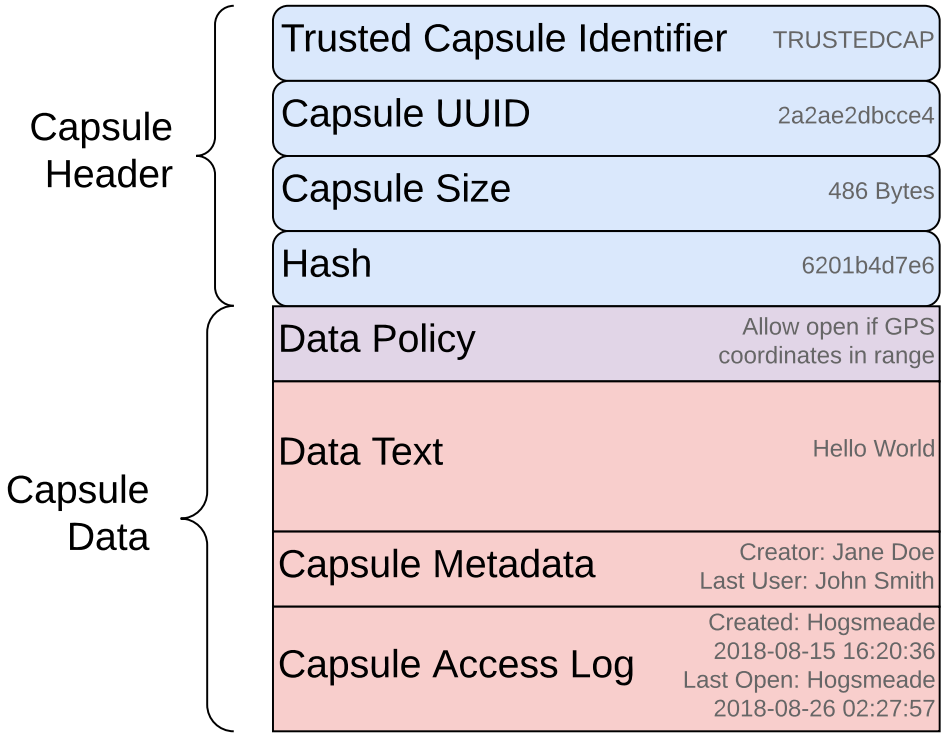

Figure 4.1: Trusted capsule layout.

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. 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-

	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

oriented APIs such as `getState` and `setState`. When using such methods, the policy must specify *where* 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 [6], it is possible to safely make an HTTPS connection from the secure world without trusting 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 are 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 actually delete the file until the file's reference count becomes zero

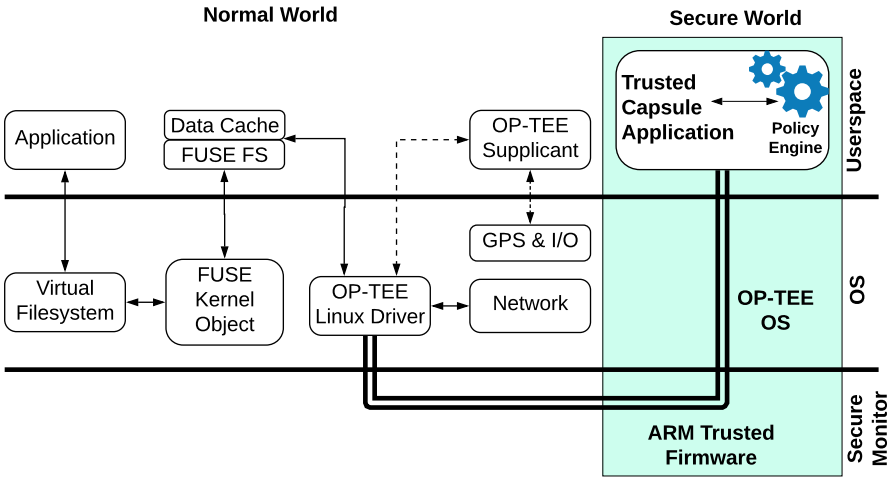

Figure 4.2: Trusted capsule data monitor design. Application system calls to the filesystem for accessing trusted capsules are intercepted and forwarded to the trusted capsule application through the FUSE filesystem and OP-TEE Linux Driver. The secure world trusted capsule applications access peripheral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

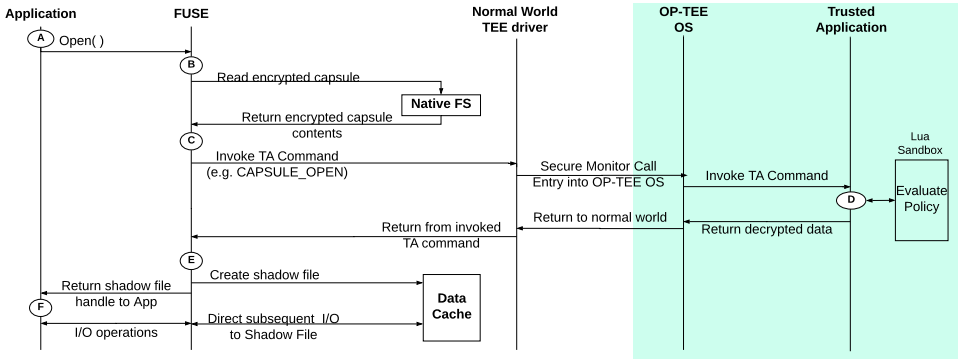

Figure 4.3: Trusted capsule monitor operation (shaded region operates in the secure world). **A.** Application *open* system call is intercepted. **B, C.** FUSE identifies if a file is a capsule, and if so, invokes an RPC into the secure world to decrypt the capsule. **D.** The trusted capsule application (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 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 within OP-TEE OS in 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 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. 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 get 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 secure storage using 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 ad-

dressed 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 for a capsule she does not have the decryption key for.

This 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 Registring 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).

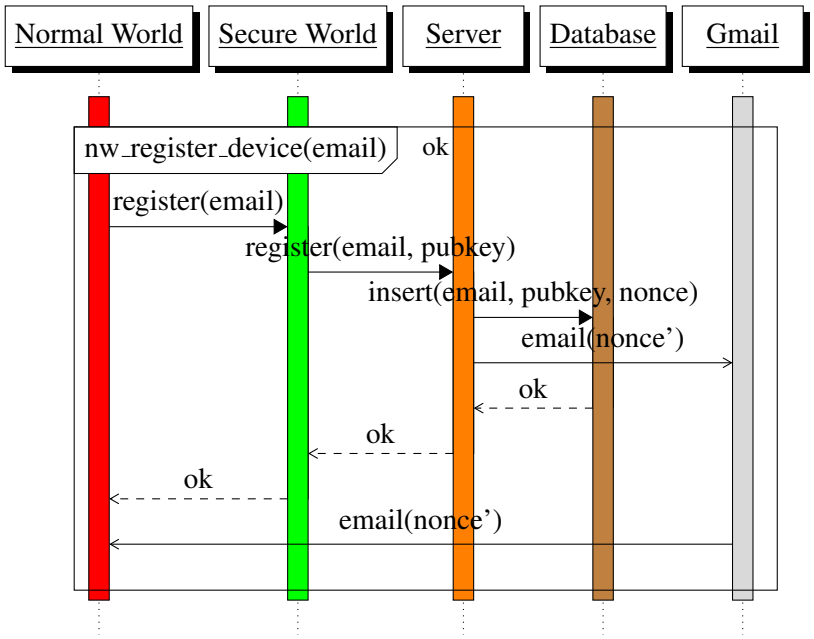

Figure 5.1: Registration as Recipient - intiating the registration process

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.

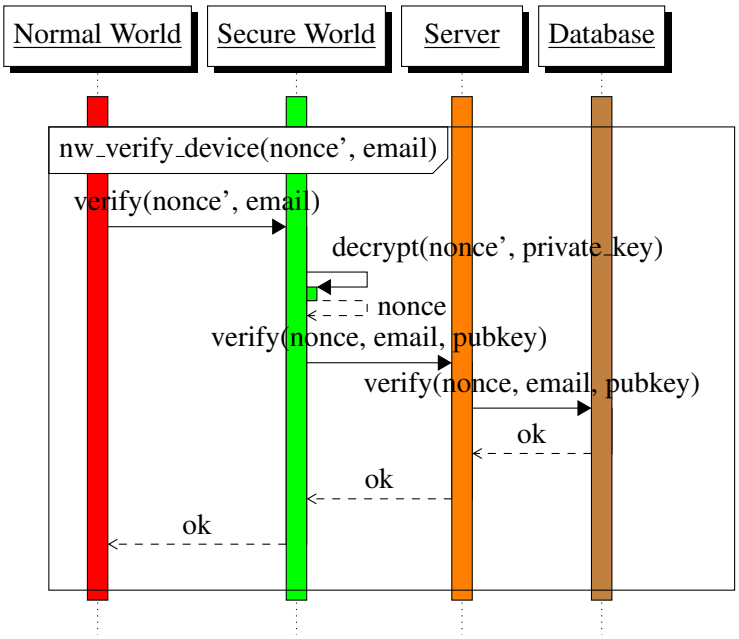

Figure 5.2: Registration as Recipient - validation of the request

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

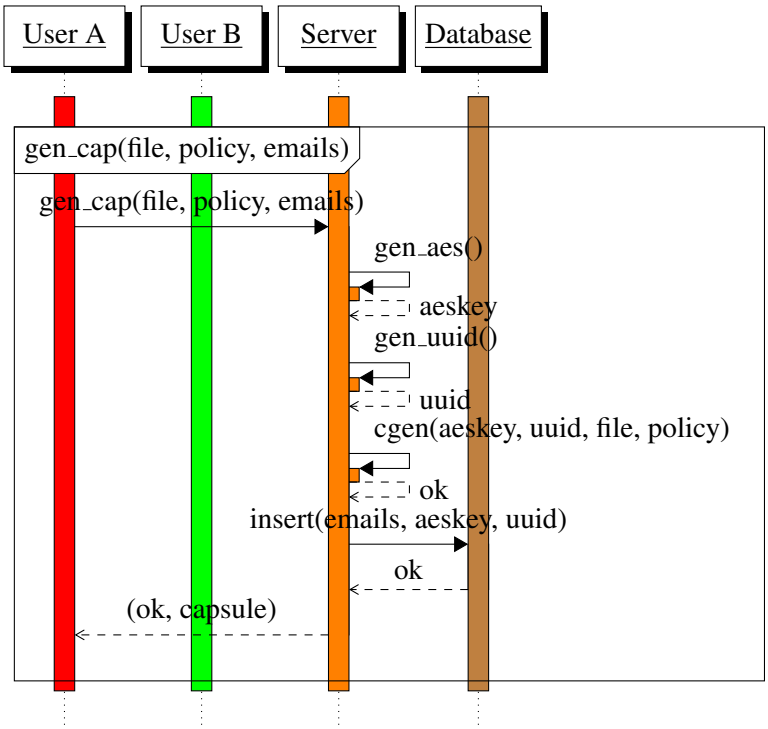

Figure 5.3: Capsule Generation and Key Registration

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

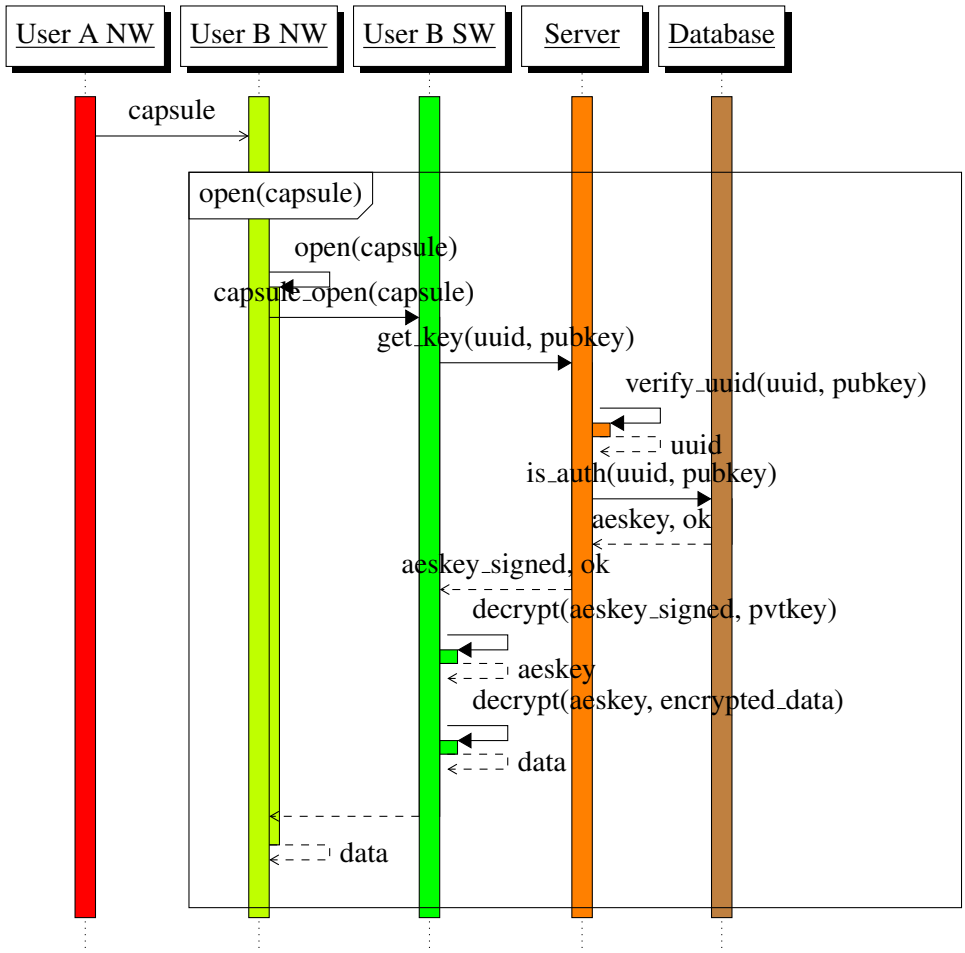

Figure 5.4: Capsule Decrypt as Recipient

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 only view shared pictures 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 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 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, consider Figure 6.2 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.

```
1  longitude = 1250
2  latitude = 200
3  range = 10
4
5  function evaluate_policy( op )
6      if op == POLICY_OP_OPEN or op == POLICY_OP_CLOSE then
7          long, lat, err = getLocation( POLICY_LOCAL_DEVICE )
8              if err ~= POLICY_NIL then
9                  comment = "Failed to getLocation"
10                 return false
11             end
12             if math.abs(long - longitude) <= range
13             and math.abs(lat - latitude) <= range then
14                 comment = "GPS coordinates in range"
15                 return true
16             else
17                 comment = "GPS coordinates are not in range"
18                 return false
19             end
20     end
21 end
```

Figure 6.1: Simple location based access policy

```
1  -- remote server information
2  remote_server = "198.162.52.127:3490"
3  -- return keywords
4  policy_result = POLICY_NOT_ALLOW
5  comment = ""
6
7  -- policy-specific keywords
8  open_time = 1523338041
9  opened = "opened"
10
11 function evaluate_policy( op )
12     if op == POLICY_OP_OPEN then
13         r, err = getState( opened, POLICY_CAPSULE_META )
14         if r == "true" then
15             return true
16         else
17             curr_time, err = getTime( POLICY_REMOTE_SERVER )
18         end
19         if err ~= POLICY_NIL then
20             policy_result = err
21             comment = "Failed to get time from remote server"
22             return false
23         end
24         if curr_time > open_time then
25             err = setState( opened, "true", POLICY_CAPSULE_META )
26             if err ~= POLICY_NIL then
27                 policy_result = err
28                 comment = "Failed to update capsule metadata"
29                 return false
30             end
31             return true
32         end
33     end
34 end
```

Figure 6.2: Policy to allow content pre-distribution

Chapter 7

Prototype

We prototyped Trusted Capsules on a LeMaker HiKey development board [4]. 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 that 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 to express complex policies such as redaction and revocation.

8.2 System call microbenchmarks

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

Policy	LOC
Location Based Access	30
Location Based Redaction	45
Content Distribution	28

Table 8.1: LOC for example policies from Section 6.

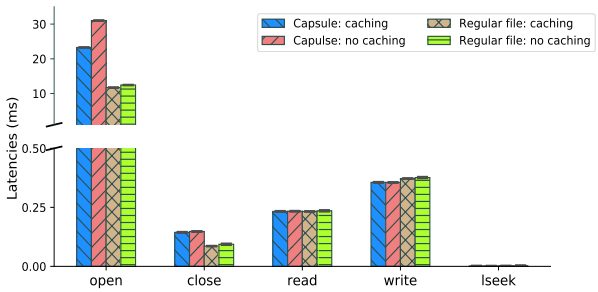

Figure 8.1: Average system call latency

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 not impacted, except for *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

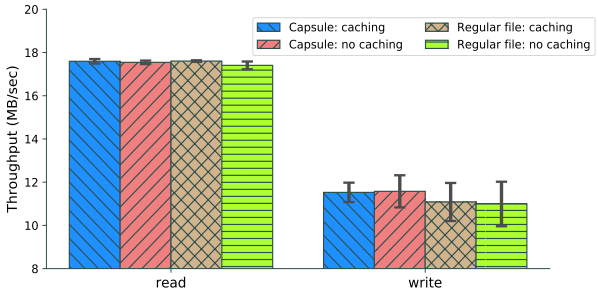

Figure 8.2: Throughput of Read and Write operations to a capsule

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

The observed altency 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 a capsule close.

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 preliminary 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

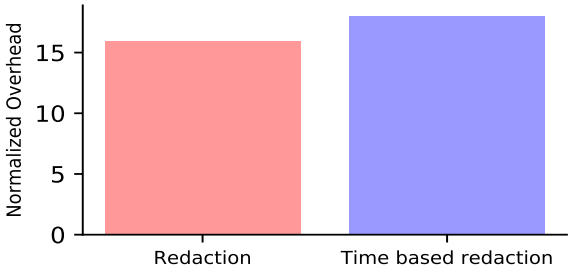

Figure 8.3: Normalized latency of servicing an `open` for different policies with respect to the latency to service a null-policy capsule open request.

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 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.

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. Moreover, 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 prohibitively impact 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 text editor, it may open the file multiple times to seek through the file in parallel. Hence, while the capsule was opened just once from a user's perspective, the policy

would observe multiple capsule access attempts. 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 damages a malicious policy can inflict. It can, for example, access a user's GPS data and send them to a server for the purpose of tracking 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 a genuine access and a 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 fulll 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 mountpoint. 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 mountpoint 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 mountpoint 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.

Chapter 10

Securing Unmodified Applications

TEEs have been around for a long time now, and have been used for a diverse range of applications as can be seen in section 11. While these bodies of research might target different problems, the way in which these projects structure their secure applications roughly falls into three distinct categories which differ based on the information being protected and the threat model around which the application has been designed:

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 [10, 38, 50, 53].

   The basic idea is that there is some easily identifiable functionality that can 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 sytem 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 crit-

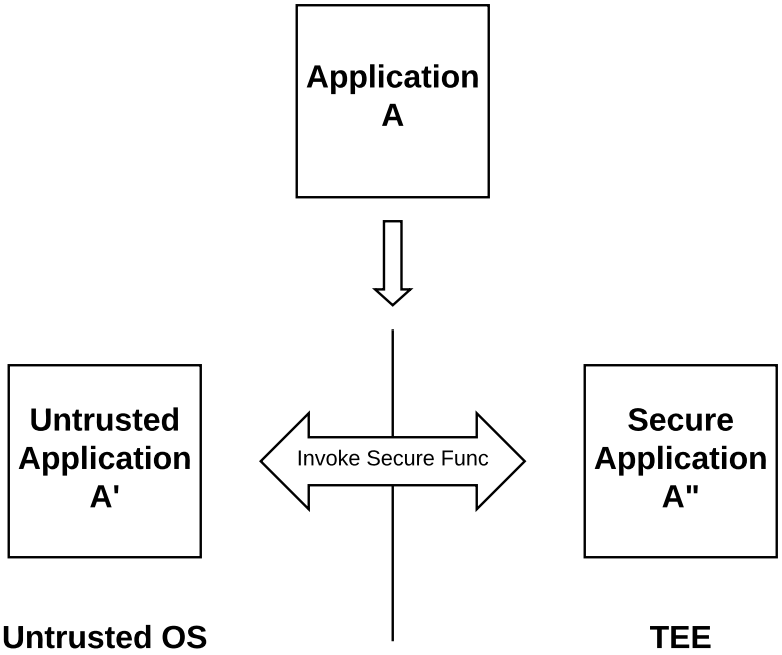

Figure 10.1: An application can be split into two parts - one that resides in the unstrusted operating system and has the interfaces to secure functionality that resides in the trusted environment.

ical functionality that needs protection is small, well defined, and can be extracted away into a TEE-resident service. Prime examples of this is 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 unmodified application.

2. **Compiler Automatically Sequester the Applcation**: There has been work in securing application using comiler wizardry. VirtualGhost [15] defines a compiler based instrumentation of the kernel and the application to protect the applications data condifentiality 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

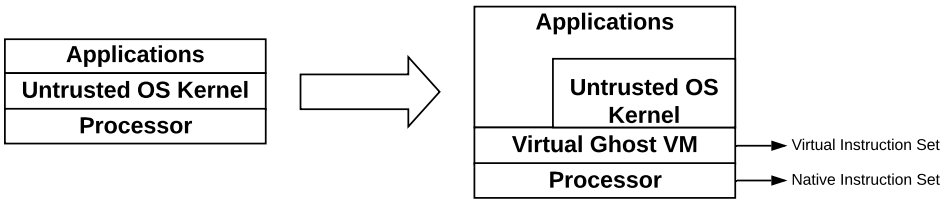

Figure 10.2: Virtual Ghost uses LLVM to create an intermediate layer the OS and the processor to protect the application from teh unfettered access otherwise enjoyed by a kernel

has into the state of the application. This is 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 actually splitting the application using the TEE API [37, 47]. This approach involves using static analysis and dataflow analysis along with optoinal annotations in the program to automatically identify partitioning scheme, and then refactoring the application into two parts that are bridged using the TEE API. Programming language theory can be used to demonstate equivalence between the split halves and the original application.

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

3. **Complete Isolation of the Application Inside the Secure World**

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.

This approach can be used in some cases, however. In Hypervision [11], the normal world kernels memory mangement is handled entirely from within TrustZone, thereby allowing opportunities for real-time monitoring of the normal world kernel. There is marginal performance deterioration in this case because there is very little code being executed in the TEE; the penalty arises from the cost to switch between the normal and the secure world. On the other hand, TrustedCapsule's first prototype was overtly 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.

All the systems presented in the related works section tend to focus on applications that rely on a rather narrow set of services from the TEE. Cryptography, SecureUI.

Chapter 11

Related Work

Securing data with policies: The concept of associating policies to data to authenticate accesses to that data is not new. An early expression of this is XACL, which specifies access control policies within XML documents [23]. Karjoth et al. proposed using *sticky policies* to provide enterprises better oversight over the customer data they collect [28]. 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 [42, 44]. 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 [40]. 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 [36]. 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 [32]. In principle, this approach allows policy execution and data access in normal world (outside TrustZone) while guaranteeing the absence of malicious applications.

Other related work 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 [49]; 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 [51]; 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 [26]; and P3, a private photo-sharing service that protects images shared by users from untrusted service providers [45].

Trusted Capsules differs from these in its aim and scope: it uses the sticky policy technique to allow end users to protect their own 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 [7] 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 [14, 17, 31, 43, 46, 55, 56]. 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 allow a process to access multiple pieces of data. Trusted capsules are less composable. If two trusted capsules have contradictory policies, they cannot be accessed by a process at the same time. 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 [19, 20, 25, 57]. It tracks information flow by interposing on the system operations at the instruction-level. These solution can track the flow of information at extremely fine granularity. 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 [8] and Hypori [5], attempt to prevent sensitive data from leaving in the first place by enforcing policy at the network boundary between external and internal systems. In these cases, policies that restrict movement of sensitive data can still be defeated by transformations, such as encryption and compression. In addition, some of these solutions incur substantial network cost as they do not support offline operations.

Finally, other work has sought to ensure data confidentiality by enforcing application structures [24, 33], limiting data lifetimes [16, 27] and providing recourse actions such as backtracing intrusions [21, 29].

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 [18, 30, 50], to provide a root-of-trust for performing runtime measurements [11–13, 52] and to secure peripherals [39]. 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 [35]; SeCloak provides direct control (on/off) over device peripherals even when the normal world OS is compromised [34]; Truz-Droid enables users to securely input and send secrets e.g., login credentials, to authorized servers without executing third-party code in TrustZone [54]; TrustShadow protects applications from untrusted OSes by executing them with a runtime in TrustZone [22]; 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 [48]; DelegaTEE, which uses Intel SGX to enable users to share their access to online service providers without revealing their credentials [41].

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 45

[2] Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware. Accessed: 2019-02-15. → page 6

[3] Global platform api specifications. http://www.globalplatform.org/. Accessed: 2019-02-15. → page 8

[4] LeMaker HiKey. http://www.lemaker.org/product-hikey-index.html. → pages 5, 30

[5] Hypori. http://www.hypori.com/. Accessed: 2019-02-15. → page 45

[6] ARM mbed TLS. https://tls.mbed.org. → page 15

[7] Sandstorm. https://sandstorm.io/. Accessed: 2019-02-15. → page 44

[8] Secure spaces. https://www.spacesmobile.com/. Accessed: 2019-02-15. → page 45

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

[10] 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 39

[11] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014. → pages 42, 45

[12] 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.

[13] 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 45

[14] 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 45

[15] 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 40

[16] 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 45

[17] 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 45

[18] 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 45

[19] 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 45

[20] 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 45

[21] 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 45

[22] 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 46

[23] S. Hada and M. Kudo. XML Access Control Language: Provisional Authorization for XML Documents. http://xml.coverpages.org/xacl-spec200102.html. → page 43

[24] R. Herbster, S. DellaTorre, P. Druschel, and B. Bhattacharjee. Privacy capsules: Preventing information leaks by mobile apps. In *Proc. of MobiSys*, 2016. → page 45

[25] 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 45

[26] 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 44

[27] J. Kannan and B.-G. Chun. Making programs forget: Enforcing lifetime for sensitive data. In *HotOS*, 2011. → page 45

[28] 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 43

[29] S. T. King and P. M. Chen. Backtracking intrusions. *ACM SIGOPS Operating Systems Review*, 37(5):223–236, 2003. → page 45

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

[31] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 321–334. ACM, 2007. → page 45

[32] A. Lazouski, F. Martinelli, P. Mori, and A. Saracino. Stateful Usage Control for Android Mobile Devices. In *Proceedings of STM '14*, September 2014. → page 44

[33] 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 45

[34] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. SeCloak: ARM Trustzone-based Mobile Peripheral Control. In *Proceedings of MobiSys '18*, June 2018. → page 46

[35] 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 46

[36] 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, 43

[37] 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 41

[38] 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 39

[39] 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 45

[40] 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 43

[41] 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 46

[42] 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 43

[43] 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 45

[44] 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 43

[45] M.-R. Ra, R. Govindan, and A. Ortega. P3: Toward privacy-preserving photo sharing. In *Proceedings of NSDI '13*, April 2013. → page 44

[46] 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 45

[47] 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 41

[48] A. A. Sani. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proceedings of MobiSys '17*, June 2017. → page 46

[49] 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 44

[50] 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 39, 45

[51] 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 44

[52] 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 45

[53] 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 39

[54] 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 46

[55] 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 45

[56] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *NSDI*, volume 8, pages 293–308, 2008. → page 45

[57] 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 45

Appendix A

Supporting Materials

This would be any supporting material not central to the dissertation. For example:

- additional details of methodology and/or data;
- diagrams of specialized equipment developed.;
- copies of questionnaires and survey instruments.