ABSTRACT

Title of Dissertation: ASSURANCE AND CONTROL OVER SENSITIVE DATA ON PERSONAL DEVICES

Matthew Lentz
Doctor of Philosophy, 2020

Dissertation Directed by: Professor Bobby Bhattacharjee
Department of Computer Science

Personal smart devices provide users with powerful capabilities for communication, productivity, health, education, and entertainment. Applications often operate over sensitive data related to the user: collecting and processing input data from sensors (e.g., fingerprint scans, location updates), or rendering output data to the user (e.g., display financial information). This sensitive data is the target of many attacks, which range from malicious applications to compromises of the platform software, which includes the operating system (OS) and privileged services. Today, users are ultimately unable to control or reason about how their sensitive data is processed, protected, or shared.

In this dissertation, I argue the following thesis: *Introducing an enforcement layer between the hardware and platform software can enable end-to-end secure applications while giving users fine-grained control over their devices.* I support this thesis through the design, implementation, and evaluation of two different instantiations of such an enforcement layer: SeCloak and AIO. SeCloak focuses on addressing a

single point in the policy space for giving control back to users: on/off control of peripherals (e.g., camera, microphone). SeCloak runs as a platform-agnostic layer that provides the abstraction of secure, virtual switches that the user can reliably configure. AIO introduces a new "accountable path" abstraction that enables constructing and reasoning about the end-to-end I/O stack between application endpoints and underlying hardware devices. Accountable paths allow for more expressive policies to be enforced over the software stack, which can be used to derive various assurances over the data (e.g., confidentiality, provenance); principals can reason about the state of the system through attestations provided by AIO over (parts of) these paths. The guarantees provided by these enforcement layers hold regardless of the correctness of the rest of the platform software (including the OS).

ASSURANCE AND CONTROL OVER SENSITIVE DATA ON PERSONAL DEVICES

by

Matthew Lentz

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Professor Bobby Bhattacharjee, Chair/Advisor
Professor Peter Druschel
Professor David Levin
Professor Neil Spring
Professor Mark Shayman

Acknowledgments

I want to extend my gratitude to everyone who has supported me in different ways during the time I have worked towards this dissertation. You know who you are, and I want you to know that I am very grateful. However, I would also like to take the time acknowledge some folks on an individual basis.

I want to start by thanking my advisor, Bobby Bhattacharjee. I was originally drawn to his approach to teaching and reasoning about problems when I took his undergraduate course on computer networks (CMSC417). After deciding to return to complete a doctorate degree, I reached out to him to see if he was willing to take me on as a student, and the rest is history. I appreciate the flexibility in his advising style, as I have been able to define and work on a number of interesting research projects across a wide variety of areas. I am very grateful for all of the time he has spent helping me to hone my skills as a researcher.

I have been fortunate to have a number of other mentors that have helped me extensively. I have had the opportunity to work with Peter Druschel throughout my Ph.D., starting with my very first research project on encounter-based networking. He has always been available to discuss my research and approaches to solving very challenging problems. Dave Levin has been someone I could continually turn to, both as a mentor as well as a friend. I have learned a great deal from his approach to writing papers and creating slide presentations that embody both form and function. I am also very grateful to Neil Spring, who has always been there to ask the tough questions and provide me with valuable feedback on my papers

and talks. Finally, I want to thank Mark Shayman as well for his feedback on this dissertation.

There are a number of friends and work colleagues who I would like to thank for improving my quality of life and for all of their help along the way (in roughly chronological order): Derek Aucoin, Jamie Jenshak, Aaron Shalev, Vasselios Lekakis (Lex), Ramakrishna Padmanabhan, Brandi Adams, Hallie Morris, James Litton, Zhihao Li, Frank Cangialosi, Andrea Bajcsy, Amelia Bateman, Katura Harvey, Stephen Herwig, and Richie Roberts.

Last but not least, I want to also extend my sincere thanks to my family. In particular, I want to thank my parents for their unwavering support and encouragement from the very beginning. They have also served as amazing role models in terms of dedication and taking pride in one's work.

This dissertation was supported by the following NSF awards: CNS-1314857 and CNS-1526635.

This dissertation involved collaborative efforts with the following people:

Chapter 4: My co-authors for SeCloak [1] were Rijurehkha Sen, Peter Druschel, and Bobby Bhattacharjee. I would also like to thank the reviewers and our shepherd Alec Wolman for their valuable feedback.

Chapter 5: My co-authors for AIO [2] were Peter Druschel, and Bobby Bhattacharjee.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Personal smart devices provide users with powerful capabilities by enabling a rich set of applications and services in areas such as communication, productivity, health, education, and entertainment. In many cases, users carry these devices around with them at all times, enabling observation of much of the user's virtual and physical life. Such applications and services often operate over sensitive data related to the user: collecting and processing input data from sensors (e.g., fingerprint scans, location updates), or rendering output data to the user (e.g., display financial information). Naturally, it is important for users to protect how their data is being collected, processed, and shared.

The controls provided to users by existing platforms for managing how their sensitive data is collected, processed, and shared are insufficient and brittle. Many high-profile attacks have exposed this insufficiency; for instance, Android doesn't require permission for applications to use the accelerometer, which could be leveraged to determine sensitive keyboard inputs (e.g., PINs, passwords) [3, 4]. The Control Center in iOS disconnects, but doesn't actually *disable*, WiFi and Bluetooth radios [5]. More importantly, even when controls are available, they are brittle because they rely on the integrity of the platform software, which is complex and

presents a large attack surface, as demonstrated by frequent data breaches [6–13].

Beyond a lack of control, users are unable to reason about the software stack that handles their sensitive data. This software stack ranges from the applications to the hardware I/O devices themselves, which always serve as the sources and/or sinks for this sensitive data. Even some hardware security features do not provide complete assurance due to their reliance on software; for instance, notification LEDs that are used to signal that the camera is in-use can be disabled by software-based attacks [14, 15]. Identifying the precise set of software components and principals responsible for handling the I/O data is challenging, as such information is not currently exposed; additionally, the monolithic nature of most software platforms significantly increases the size of the computing base included in the set. Finally, it is not possible to determine the assurances that the platform provides over the I/O data, such as confidentiality and integrity protections.

These problems actually extend beyond the end users themselves to other stakeholders in the ecosystem. For example, consider a mobile banking application that allows users to issue monetary transactions. The bank may require assurance that the user explicitly confirmed the transaction prior to executing it; likewise, the user desires the assurance that the amount they entered is actually used in the transaction. If the OS, or privileged software for managing the display and keyboard, is compromised, these assurances may not hold; for instance, a compromised OS could issue fake inputs to the bank application to change the amount or the recipient of the transaction. The status quo leaves both the user and the bank without recourse: both must accept the facilities provided by the platform software and

trust that they are correct and not compromised.

Existing mechanisms that provide protections for sensitive data rely on augmenting existing platforms with support for trusted execution environments (TEEs). These TEEs support running isolated, trusted applications (TAs) and provide secure access to hardware I/O devices. Commercial solutions [16–19] provide building blocks for applications, such as PIN input and biometric authentication primitives. Prior research has focused on supporting trusted I/O channels to specific devices [20–31], and providing supporting generic mechanisms for constructing channels to arbitrary devices [32–35]. Unfortunately, such solutions 1) require the adoption of an all-trusted security principal that supplies the complete software stack, 2) prevent other principals from extending (or replacing) the secure functionality, and 3) do not provide mechanisms for control over the secure functionality.

1.1 Thesis

> *Introducing an enforcement layer between the hardware and platform software can enable end-to-end secure applications while giving users fine-grained control over their devices.*

I use *platform software* to refer to the operating system (OS) and any privileged services or frameworks that run on top of the OS. I use the term *users* in this statement to refer to both end users themselves, as well as other stakeholders in the ecosystem (e.g., device manufacturers, platform providers, application principals). Providing users with *fine-grained control* means that users should be able express

policies independently from the rest of the platform software, and that these policies can be applied to any portion of the I/O stack.

My work in this dissertation focuses on defining the abstractions and mechanisms for the enforcement layer. The enforcement layer must be isolated from the (untrusted) platform software and have the ability to mediate software accesses to the underlying hardware devices. To achieve these properties, the enforcement layer relies on enabling hardware mechanisms such as virtualization and/or security extensions. I provide a background on these mechanisms in Chapter 2, including a detailed overview of ARM TrustZone security extensions which I leverage in prototype implementations of my own enforcement layers.

In addition to the primary goals for the enforcement layer defined in the thesis statement, there are other secondary goals to consider. First, the enforcement layer must be *efficient*, since many I/O operations require low-latency accesses and some involve transferring large amounts of data (e.g., network, display). When no policies are applied to a given I/O device, the overhead should be negligible. Second, the enforcement layer should provide mechanisms that help promote *code reuse*, such as from the existing platform software; applications should only need to provide their own implementations of components when explicitly required by trust assumptions. Third, the enforcement layer should support expressive policies that can be enforced at any layer of the I/O stack, thereby decoupling policy from the mechanisms used to mediate I/O accesses.

1.2 Contributions

To support my thesis, I have designed, implemented, and evaluated two systems that act as enforcement layers: SeCloak (Chapter 4) and AIO (Chapter 5). In my initial work on SeCloak, I focused on supporting a single, yet important, point in the policy space for giving users control over their devices: on/off control of I/O devices. Building off my experience from SeCloak, AIO sets out to provide a single abstraction of an "accountable path", which enables end-to-end secure applications with first-class support for expressive policies. This dissertation is structured as follows:

Chapter 2: Background and Related Work

I provide a background on hardware mechanisms and software techniques to enable trusted and isolated execution, which our enforcement layer will require. I discuss existing work and place it into context with respect to the contributions of my own work. This discussion focuses on two broad categories of work: 1) support for enforcing policies over I/O control and data, 2) enabling trusted channels to securely interact with I/O devices, and 3) relevant OS architectures.

Chapter 3: Enforcement Layer

I motivate the introduction of an isolated enforcement layer by discussing different possible solutions and their issues. I discuss the goals for such an enforcement layer in more detail, and then describe the trust and threat model that I consider.

Chapter 4: SeCloak: On/Off Control for Peripheral Devices

I describe SeCloak [1], an enforcement layer designed to support a single, yet

important, point in the policy space: on/off control of peripheral I/O devices. For example, journalists interviewing confidential sources may want to use the microphone but reliably turn off the radios and GPS, and users may want to make sure the camera and microphone are disabled during private meetings. SeCloak provides a simple abstraction to users of secure virtual switches that they can use to control the various peripherals on their device (e.g., microphone, camera). SeCloak acts as a separate, platform-agnostic layer that can enforce the user's policy settings even if the rest of the system software, such as the OS and applications, are compromised.

Chapter 5: AIO: Control and Assurance Over Sensitive I/O Data

I describe the design of AIO, an enforcement layer meant to enable transparency and expressive, end-to-end control over the software that collects, processes, and shares I/O data. AIO implements a new "accountable path" abstraction that resolves an impedance mismatch between the assurances and control desired by users (and other principals) and the mechanisms provided by existing platforms. Accountable paths handle multiple (mutually distrusting) principals, enabling fine-grained delegation of trust and safe composition of software from principals that may contribute different portions of the I/O stack. Accountable paths enable more-privileged principals, such as the end user, to enforce policies over less-privileged software components. These policies can be used to obtain various forms of assurance over I/O data, such as confidentiality and provenance; AIO provides a way to prove these assurances to principals through (remote) attestation.

Chapter 6: Conclusion and Future Work

I conclude by revisiting the contributions of this work. I discuss future work

first in terms of open technical challenges related to this work, namely with respect to usable security and strengthening the foundations of trust in the enforcement layers (SeCloak and AIO) that I presented. I also discuss future work in terms of non-technical challenges related to the practical deployment of this work on contemporary devices.

Chapter 2: Background and Related Work

In this chapter, I begin by providing a background in trusted and isolated execution, both in terms of enabling hardware mechanisms the software systems that are built on top. Next, I discuss related work that falls into three categories: 1) support for enforcing policies over I/O control and data, 2) enabling trusted channels to securely interact with I/O devices, and 3) relevant OS architectures. I put this prior work into context with respect to my work on SeCloak and AIO.

2.1 Trusted and Isolated Execution

The enforcement layer relies on enabling mechanisms from the underlying hardware to: 1) isolate itself from all other software running on the system, and 2) mediate software-to-software and software-to-hardware interactions. Next, I will describe hardware technologies and associated software systems that can be used to achieve similar isolation and mediation properties. In particular, I focus on the ARM TrustZone security extensions which serve as the basis to enable my implementations of SeCloak and AIO (see Sections 4.4 and 5.4 respectively).

2.1.1 Virtualization Extensions

Hardware extensions for virtualization are provided by all the major CPU vendors: Intel [36], AMD [37], and ARM [38]. Virtualization of the memory management unit (MMU) allows virtual memory of the host to serve as physical memory of the guest. While these mechanisms handle memory accesses made by software running on the CPU, they do not support virtualization of the memory accesses made by DMA-capable I/O devices. Instead, input-output memory management units (IOMMUs) are employed map device addresses to physical addresses. IOMMUs allow scattered memory pages to be treated as contiguous by the device, enable devices to work with smaller address ranges than that of the system bus, and also (importantly) provide protection against maliciously configured devices.

Many systems use hypervisors to protect secure applications from vulnerable OSs. Overshadow [39] employs a hypervisor to protect applications from the untrusted OS through the use of multi-shadowing, where the hypervisor selects the shadow page table to use based on the visible context (e.g., protection ring, instruction pointer). The untrusted OS is able to manage the memory resources of applications, but only gets an encrypted view of user pages. The application itself is able to operate on an unencrypted view of its pages. Inktag [40] and Sego [41] additionally allow the isolated applications to safely make use of untrusted (but verified) OS services for file storage and handling crash recovery. TrustVisor [42] is a hypervisor that protects the integrity of code, as well as confidentiality and integrity of data, for security critical portions of an application. TrustVisor presents

a "micro-TPM" interface to applications, providing functionality such as attestations and sealed storage that runs efficiently on the CPU as opposed to traditional hardware TPM devices.

2.1.2 Security Extensions

Beyond virtualization, modern architectures include hardware security extensions, such as Intel SGX [43], Sanctum [44] and ARM TrustZone [45]. These techniques go beyond Trusted Platform Modules (TPMs) that enable secure boot, or Intel Trusted eXecution Technology (TXT) [46] and AMD Secure Virtual Machine (SVM) [47] that allow for attested execution of the OS or smaller code segments [48].

2.1.2.1 Intel SGX

Intel SGX [43] enables on-demand "enclaves" which protect the execution of the software within against both hardware attacks (e.g., probing on the memory bus) and software attacks (e.g., compromised OS or hypervisor). SGX attests to correct execution via measurements of the initial state of the enclave (code and data), and protects the confidentiality of enclave memory by transparently encrypting writes (and decrypting reads) to (from) main memory. SGX enclaves have been used to securely run existing binaries without modification by the OS [49, 50], and have also been extended [51] to support *untrusted* computation on sensitive data that involves multiple parties.

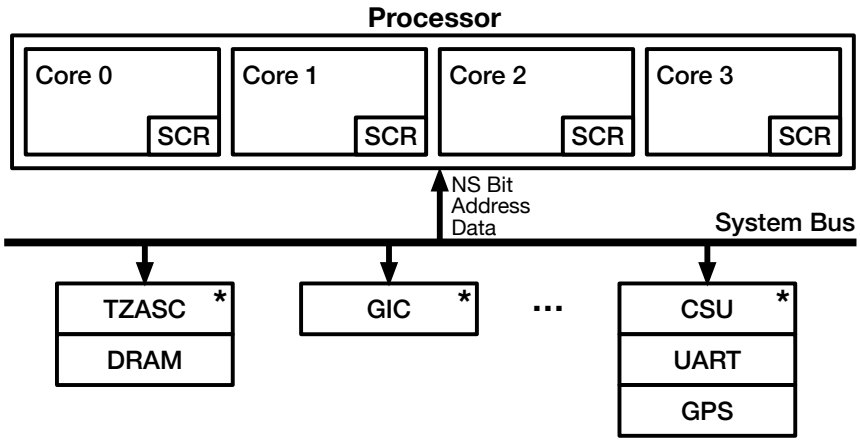


Figure 2.1: High-level overview of ARM TrustZone SoC architecture.

2.1.2.2 ARM TrustZone

Figure 2.1 shows a high-level overview of the ARM TrustZone architecture, with a particular focus on relevant security components.

ARM TrustZone [45] is a set of hardware security extensions that supports hardware isolation of two "worlds" of execution: non-secure and secure. Each processor core executes in the context of a single world at any time; the Security Configuration Register (SCR) contains the "NS" bit that denotes the current world state of the processor core. In each world, the processor core may execute in different privilege levels (PLs) in ARMv7 [52] or exception levels (ELs) in ARMv8 [53]; the following discussion will assume the ARMv7 architecture. Both the non-secure world and secure world may execute in PL0 and PL1, which correspond to user mode and privileged modes (e.g., supervisor, IRQ, FIQ) respectively. The non-secure world may execute in PL2, which corresponds to hypervisor mode. The secure world PL1 also contains a monitor mode, which is used for executing a secure software monitor (discussed next).

A software monitor, installed by the secure world, carries out all transitions between worlds. These transitions are either explicit via the `SMC` instruction invoked by non-secure supervisor-mode code, or implicit due to an access fault or secure interrupt. The `SMC` instruction acts as a call gate mechanism for the non-secure world to invoke a particular secure-world function. ARM provides a standard calling convention [54] when invoking the `SMC` instruction that specifies the service (e.g., CPU, OEM, Trusted App) and an identifier for a function that belongs to the service. The SCR register that contains the current world state also has a "EA" bit that determines the behavior of external aborts, such as those caused by hardware protection access faults. This bit controls whether the processor will switch to abort mode or monitor mode on receiving the external abort exception. Similar bits exist for IRQ and FIQ modes to configure interrupt behavior.

All accesses to memory and I/O devices are tagged with the "NS" bit, which specifies whether the access was issued while the core was in non-secure mode. This tagging enables hardware components, such as bus and memory controllers, to be TrustZone aware and support access control based on the world performing the access; we show several such devices in Figure 2.1. The TrustZone Address Space Controller (TZASC) [55] mediates accesses to RAM, and can be configured to allow or deny accesses to regions according to the world state and type of access (R/W). The secure world OS can use the TZASC, for instance, to protect the parts of RAM it uses from read or write accesses by the non-secure world. The Generic Interrupt Controller (GIC) [56] is a TrustZone aware device that enables the secure world OS to assign specific interrupts as either secure world or non-secure world, and supports

limiting the priority of non-secure world interrupts (e.g., to avoid DoS attacks).

There are also manufacturer-specific IP blocks that are used to protect access to peripheral devices, such as display controllers and peripheral bus controllers. While these blocks vary in their exact instantiation from manufacturer to manufacturer, the functionality they provide is similar; in the following discussion, I will focus on NXP's Central Security Unit (CSU) block that they include in their SoCs. The Central Security Unit (CSU) allows the secure world to configure access control in two ways. First, the CSU provides settings to allow or deny accesses to devices connected to the system bus based on the world state, mode state, and type of access (R/W). Optionally, when the access is denied, the CSU can be configured to deliver the fault to the secure world monitor to be handled. Second, the CSU provides settings to determine the world state used to tag non-CPU, DMA-capable devices' accesses on the system bus. This setting can be used to constrain the accesses of DMA-capable devices that are managed by the non-secure world to prevent the devices from reading (or modifying) the secure world memory.

In order to provision the system in a secure way, TrustZone provides support for a secure boot chain [57]. A hardware root of trust, in the form of hash of a public key stored in secure, one-time programmable (OTP) fuses, is used to establish the chain of trust. A first-level boot ROM stored on the SoC executes after the device is powered on (or reset). This boot ROM is responsible for loading and verifying a second-level boot loader that is stored in external storage; the verification is performed by checking the signature of the boot loader image against the public key attested to by the hardware root of trust. The boot loader will subsequently

load and verify the secure world OS, using either the same public key or a new one embedded within the image of the boot loader itself. Once the secure world OS executes, it can configure all of the hardware appropriately before subsequently loading and executing the non-secure world boot loader and OS.

Unlike SGX, which supports an arbitrary number of enclaves, TrustZone supports a single trusted/isolated execution environment (i.e., the secure world). Komodo [58] provably extends TrustZone to support attested isolated execution environments similar to Intel SGX enclaves (though physical memory snooping remains out of scope.) While SGX is implemented entirely in hardware, Komodo instead uses a formally verified software implementation that runs on top of ARM TrustZone. Cho et al. [59] explore a hybrid approach to supporting isolated execution environments with both a hypervisor and ARM TrustZone. During the lifetime of a secure application, the hypervisor is active and provides isolation; otherwise, the hypervisor is disabled (reducing overhead) and TrustZone hardware protections are used to protect sensitive memory regions. Several commercial products [16,17,19,60] also implement support for isolated execution of trusted applications (TAs).

2.2 Policy Enforcement

In this section, I discuss prior work that enables policy enforcement. Motivated by the security and privacy problems in mobile devices, recent work envisions many different solutions including novel permission models [61–67] and sandboxing applications [68–71]. Reference monitors and security kernels [72–79] provide

fine-grained access-control mechanisms that can contain application misbehavior. However, all of these solutions assume that the OS itself hasn't been compromised by an attacker (or isn't malicious). In the remainder of this section, I will focus on approaches that rely on various hardware extensions described in the previous section to implement isolated software components that enforce various policies, with a particular focus on those that pertain I/O devices.

Several prior systems investigate binary (on/off) control policies for I/O peripheral devices on mobile systems [80–82]. Brasser et al. [80] support restricted spaces where usage of certain devices is not allowed. A local, isolated policy enforcement service running in the secure world on ARM TrustZone grants a remote policy server RW access to system memory; a trusted vetting server protects against rogue accesses issued by the policy server. Santos et al. [81, 82] allow users to grant "trust leases" to applications that let the them restrict usage of certain devices until a terminal condition is met (e.g., after X hours). These systems are closely related to my own work on SeCloak [1], which I discuss in more detail in Chapter 4. SeCloak does not make any assumptions about the trustworthiness or correctness of the platform OS, and can reliably secure peripheral devices even when the platform OS (and the rest of the software) is compromised.

Other systems support different points in the policy space for I/O peripheral devices beyond on/off control. Ditio [83] employs a combination of a hypervisor and a kernel in the ARM TrustZone secure world to efficiently audit sensor activities by recording logs that are later processed by a formally verified auditing tool. Viola [84] enables custom, per-peripheral notifications whenever the I/O peripheral device is

being used; for example, blinking the notification LED when the camera is active. Viola employs formal verification techniques to provide guarantees that the user will be notified. All of these policies can be implemented on top of AIO, as discussed in Chapter 5.3, which also supports composing policies at multiple privilege levels from many (mutually distrusting) principals.

BitVisor [85] is a hypervisor that enforces security on both I/O control and data accesses, mediating specific accesses to enforce security policy. While BitVisor does support arbitrary policies, there are several downsides compared to AIO since BitVisor: 1) does not isolate policies from one another or from the hypervisor itself, 2) only allows application of a single policy per device (restricting composition), and 3) does not enable policies issuing their own I/O accesses (limiting expressiveness).

Cox et al. [86] argue for the utility of running hypervisors beneath the existing platform software on commodity mobile devices. In particular, this would be beneficial for enabling secure operating systems and supporting various security services. Both SeCloak and AIO are examples of this vision for a security-focused layer.

2.3 Trusted I/O

In this section, I discuss prior work that supports secure interactions with I/O devices, which falls into one of the two following categories. First, I describe prior work that provides *generic mechanisms* for establishing trusted I/O channels to arbitrary devices. Second, I present prior work that enables secure interactions with *specific devices*, such as the display or biometric sensors.

2.3.1 Generic Mechanisms

Zhou et al. [32] present a hypervisor-based design for enabling trusted I/O channels for commodity x86 processors by leveraging virtualization extensions. The hypervisor hosts both the untrusted OS as well as the trusted program endpoints (PEs) in separate VMs. The PEs contain all of the necessary application logic and lower-level device drivers to interact with the device. In follow-up work [33], Zhou et al. address bloating of the trusted-computing base (TCB) as a result of needing to include drivers within the PEs. They introduce a trusted "wimpy" kernel beneath the application PEs, which outsources (and subsequently verifies) certain parts of the I/O stack to the untrusted commodity OS. In this work, they focused specifically on the USB subsystem, where the bus enumeration and power management functions are outsourced to the commodity OS.

DriverGuard [34] protects I/O data flows by using a hypervisor to enforce that only privileged code blocks (PCBs) can operate on the raw, unencrypted I/O data. SGXIO [35] posits a system in which a hypervisor hosts the untrusted OS (running in a VM) as well as the trusted I/O drivers (running in SGX enclaves). Applications run their trusted components inside an enclave and establish trusted paths to the necessary I/O drivers. Lacuna [87] ensures that I/O flows can be securely erased from memory once they terminate, by relying on virtualization, encryption, and direct NIC access.

As compared to these prior approaches, AIO focuses on providing support for building trusted paths via composable modules, while enabling sharing of devices

(if desired) amongst multiple non-secure and secure applications that is enforced by the credential model and first-class support for policies. AIO adopts a similar view on TCB reduction to the work by Zhou et al. [33] (i.e., re-using untrusted code), but allows for more flexibility beyond the singular "outsource and verify" approach; for example, AIO policies can be used to enforce a behavior specification over the accesses made by the untrusted code.

2.3.2 Specific Devices

Much prior work [20–31] focuses on enabling specific trusted paths such as a secure virtual keyboard, confidential display, and trusted sensors. TrustUI [20] enables trusted paths without trusting device drivers by splitting drivers into an untrusted backend and trusted frontend that runs within the secure kernel. TrustUI uses ad-hoc techniques, such as randomizing keys on the on-screen keyboard after every touch, for ensuring that the information available to the non-secure kernel does not leak device data. ShrodinText [21] is a system for displaying text while preserving its confidentiality from the untrusted OS. ShrodinText establishes a secure path between a remote (trusted) server and the local framebuffer/display for this purpose, relying on TrustZone and a hypervisor (for MMU and IOMMUs) to secure parts of the rendering stack. Liu et al. [22] uses trusted device and bus drivers, implemented in TrustZone and hypervisors, to attest and encrypt sensor readings. Finally, many of the commercial TEE products [16, 17, 19] implement support for specific trusted I/O channels, such as a secure PIN input and fingerprint-based

biometric authentication.

AIO can support the same functionality as these systems within a more general architecture, while simultaneously providing first-class support for user policies. This general architecture also supports composing pieces of functionality and policies from multiple (mutually distrusting) principals, in contrast to the traditional model in which there is a single, all-trusted security principal that is responsible for providing the secure functionality (e.g., trusted I/O channels and device drivers).

2.4 OS Architectures

Many widespread OSs, such as Linux and Windows, exhibit a monolithic structure, whereby most services (e.g., device drivers, network stack) are implemented within the OS kernel itself. While there are many benefits such as functionality and performance, there are also drawbacks to such an architecture. One such drawback is a lack of isolation between services within the kernel. This is especially important with respect to device drivers, as they are a significant source for bugs in OS kernels [88, 89]. To account for for this, there has been a shift towards implementing drivers in userspace [90, 91] for better fault isolation. Another drawback is that, due to the structure of the kernel, the size of the TCB is extremely large and thus very susceptible to vulnerabilities [92]. There has been work on leveraging hardware virtualization and security extensions to support isolation for applications that run on an untrusted host OS [39–41, 43]. Additionally, extending the kernel with new functionality is challenging, because the code that implements the functionality must

run in the same protection domain as the kernel itself; SPIN [93], VINO [94], and eBPF [95, 96] support safe extensibility. AIO leverages eBPF for safe extensibility by allowing policies to be executed within AIO's protection domain for improved performance.

In part motivated by attacks against monolithic OSs [92], microkernel architectures aim to minimize the functionality within the kernel itself and instead push as much of this functionality into userspace services as possible [97]. Many modern microkernels exist, such as seL4 [98], which has formally verified correctness and security properties, and Zircon [99], which is used as the kernel within Google's Fuchsia OS [100]. In contrast to monolithic kernels, microkernels offer far more isolation and a significant reduction in the size of the TCB. While necessary, these properties are not sufficient alone. Our design and implementation of AIO on ARM TrustZone is similar to that of a microkernel: AIO isolates each module in userspace (like services), mediates communication between modules, and exercises access control over system resources. AIO provides a way to extend (or replace) functionality provided by services, and apply and enforce policies over communication between different modules (services); AIO uses capabilities to safely compose these software components that may be associated with many mutually distrusting principals. Additionally, AIO enables reasoning about the components that make up the software stack between the underlying hardware devices and a (remote) application endpoint via attestations over AIO, modules, and the state of paths. Note that it is possible to implement AIO as a (privileged) service on top of a trusted microkernel such as seL4 [98].

Scout [101] introduces paths as a first-class abstraction within the OS. A path in Scout consists of a sequence of routers, each implementing some functionality behind a well-defined interface; the routing decisions may either be static at compile time or dynamic at runtime (e.g., demultiplexing ports for UDP/TCP packets). Escort [102] extends Scout to provide for various security features, such as isolation between (parts of) paths and supporting static filters for the interactions between routers. Unlike Scout/Escort, which require the routing graph to be defined at build time, AIO allows both modules and policies to be installed and removed at runtime. As part of AIO, we recognize the need to support multiple mutually distrusting principals that may contribute software components that make up the paths; AIO leverages a credential scheme, backed by a trusted specification of the hardware, to safely compose the modules into paths (and apply policies along the path). Finally, AIO provides stronger guarantees by isolating itself (along with all modules and policies) from the untrusted platform software, and allows for several different ways to decompose trust that allows for trade-offs between TCB size and performance.

Chapter 3: Enforcement Layer

In this chapter, I motiviate the need for an isolated enforcement layer by revisiting the goals and examining different potential solutions (and their issues). Next, I describe the trust and threat model that I assume in this work. Finally, I discuss the (trusted) hardware specification that all enforcement layers require to securely mediate access to hardware devices.

3.1 Motivation

As discussed in the thesis statement, we consider two primary goals for the enforcement layer. First, the enforcement layer should reliably enforce expressive policies that are applied by users (and other stakeholders) to the collection, processing, and sharing of I/O data. We address this problem at the level of I/O data since hardware I/O devices always serve as either the sources and/or sinks for sensitive data, whether collected via input sensors (e.g., audio, location) or sent to output devices (e.g., display, network). Second, the enforcement layer should expose the state of the device in a simple, unambiguous manner. This state should enable reasoning about the software stack between the (trusted) hardware devices and ap-

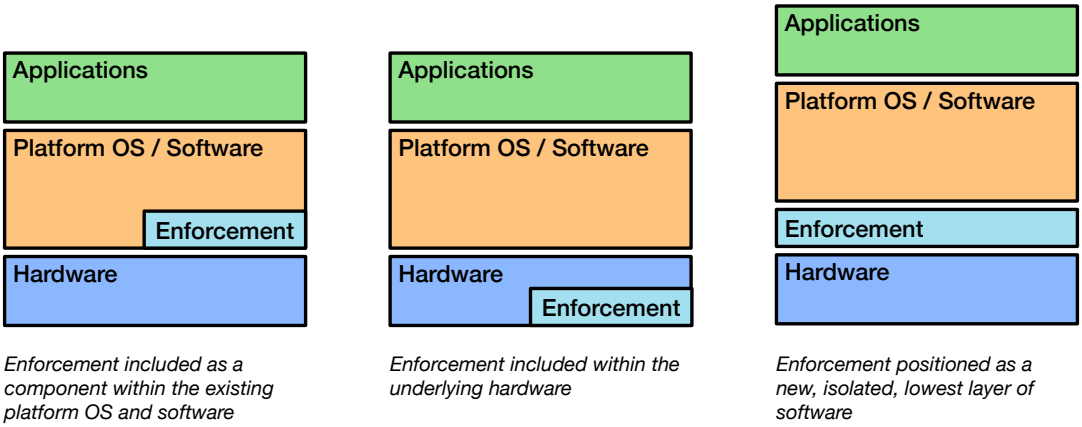


Figure 3.1: Different ways to place the enforcement component within the existing system architecture.

plication endpoints. In particular, reasoning about the assurances that this software stack provides over the data (e.g., confidentiality, integrity) and the precise set of components that make up this software stack.

There are several possible ways to augment existing systems with support for such an enforcement layer, as shown in Figure 3.1. First, we could consider placing the enforcement layer within the platform software itself (e.g., the platform OS). This would allow the enforcement layer to have high visibility into the rest of the system, as the platform software sits directly between the applications and the underlying hardware. However, as discussed earlier, the platform software has a large attack surface and has continually proven to be vulnerable to attacks; it is very hard to completely secure such a large, complex software base.

Alternatively, we could consider placing the enforcement layer within the hardware itself. This would limit the visibility of the enforcement layer, but puts it in the best possible position to mediate software accesses to hardware devices. Unfortunately, the inflexibility of the hardware layer presents a significant challenge.

Technologies, such as Intel SGX [43], that implement most of the functionality within the hardware have run into these issues as a result of new attacks and changing feature requirements. Ideally, the hardware should provide simple, generic primitives that enable various implementations and abstractions in the software layer.

Instead, we could consider placing the enforcement layer within a new, isolated layer of software that sits between the platform software and the underlying hardware. While the enforcement layer suffers the same issue of limited visibility as in the hardware approach, we gain increased flexibility by defining it within a software layer. The enforcement layer will rely on more minimal hardware support in the form of enabling mechanisms that allow the enforcement layer to: 1) isolate itself from other (potentially compromised) software on the device, and 2) mediate interactions between software and the hardware resources. By operating as an independent software layer, the enforcement layer avoids fate-sharing with the platform software as a result of attacks; however, it is important to minimize the TCB of the enforcement layer to limit the attack surface of this critical component itself.

As discussed in Section 2.1, these mechanisms may be supported by (a mix of) virtualization extensions and security extensions. For example, the facilities provided by TrustZone are sufficient for the isolation and mediation properties required to implement such an enforcement layer. Isolation can be achieved by running the enforcement layer in the secure world, with the platform software and applications running in the non-secure world. Mediation can be achieved by configuring the hardware protections and trapping access faults within the enforcement layer in the secure world.

3.2 Trust and Threat Model

There are many principals that make up the ecosystem for smart devices: device manufacturer, platform provider, user, and various application principals. A device manufacturer is responsible for building and provisioning the hardware device; Samsung is an example of a device manufacturer. A platform provider supplies the platform software, which includes the OS and privileged system frameworks and services; Google and Samsung are both examples of platform providers for the Android/Linux software platform. A user is an owner of a device; we entrust the user with full privileges over their device, which they can (temporarily) delegate to other principals. An application principal provides applications that the user installs and executes on their device; banks and health authorities are examples of application principals. These principals may be mutually distrusting.

All principals trust in the correctness and security of the enforcement layer. All principals also trust the underlying hardware devices, which includes all resources within the system-on-chip (SoC) as well as peripheral devices; all static firmware and microcode executed by the devices are included in this trusted scope as well. All principals trust that the device manufacturer provides a correct and complete specification of the hardware that they provisioned. I discuss these hardware specifications in more detail later in this chapter (see Section 3.3). Finally, all principals trust that the boot ROM provisioned by the device manufacturer is secure and correct.

The enforcement layer does not place any trust in the software from the plat-

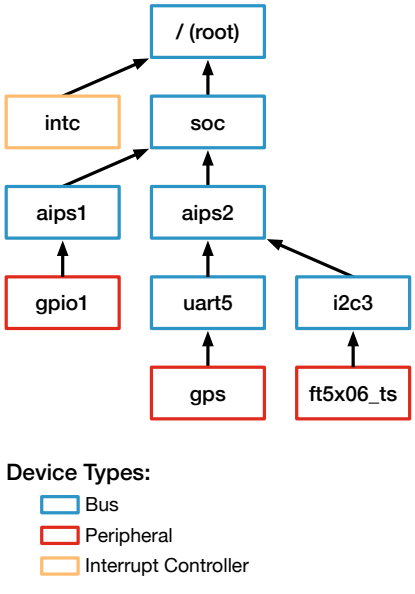
form provider, including the OS kernel, device drivers, kernel modules, and any privileged system frameworks and services. Additionally, the enforcement layer does not trust the applications that run on top of the platform software base. All of this software may be faulty, malicious, or compromised by an attacker.

I will later refine this threat model within the precise context of both SeCloak and AIO enforcement layers in their respective chapters.

3.3 Hardware Specification

The enforcement layer must understand the configuration of the hardware in order to mediate software accesses to devices. We assume the enforcement layer has access to a trusted hardware specification that describes: 1) the complete set of hardware resources on the device, and 2) how the hardware resources are interconnected. It is important that the specification provides a complete and accurate description of the hardware, since otherwise there may be other peripherals in the system that can violate any and all security properties (as the enforcement layer will not be aware of their existence).

While this specification may be written in a variety of different ways, we rely on the device tree (DT) [103] format to serve as the basis for the specification. The DT format is widely used to specify the hardware resources for embedded systems. In Figure 3.2, we present an excerpt from the DT for the Nitrogen6Q development board that contains an i.MX6 (SoC). This excerpt from the DT focuses on two devices and their dependencies: the GPS receiver device (gps) and the touchscreen




```
/ {
  intc {
    reg = <0x00A01000 0x1000>, <0x00A00100 0x100>;
  };
  soc {
    interrupt-parent = <&intc>;
    aips1 {
      gpio1 {
        compatible = "fsl,imx6q-gpio";
        reg = <0x0209C000 0x4000>;
        interrupts = <0 66 IRQ_TYPE_LEVEL_HIGH>;
    };};
    aips2 {
      reg = <0x02100000 0x100000>;
      uart5 {
        compatible = "fsl,imx6q-uart";
        reg = <0x021F4000 0x4000>;
        interrupts = <0 30 IRQ_TYPE_LEVEL_HIGH>;
        gps {
          compatible = "GP-20U7";
      };};
      i2c3 {
        compatible = "fsl,imx6q-i2c";
        reg = <0x021a8000 0x4000>;
        interrupts = <0 38 IRQ_HIGH>;
        ft5x06_ts {
          compatible = "ft5x06-ts";
          reg = <0x38>;
          interrupts-extended = <&gpio1 9 IRQ_TYPE_LEVEL_HIGH>;
      };};
};};};
```

Figure 3.2: Visualization of the device tree with a corresponding excerpt from the device tree file for the Boundary Devices Nitrogen6Q board with an i.MX6 SoC.

device (ft5x06_ts).

The DT is specified hierarchically, which follows from the layout of the hardware: each platform or peripheral device (leaf node) connects to the CPU through one or more bus devices (inner nodes). The root node (/) in the hierarchy corresponds to the system bus. Each device node in the DT is assigned a name which, along with the full path, may be used to reference the device (e.g., "/soc/aips2/uart5/gps" for the GPS receiver). Each device node contains a set of key-value properties that describe device-specific information. Some of these properties specify resource dependencies of the device on other devices in the system; the *reg* property specifies how the device may be addressed through its parent device, and *interrupts* specifies the interrupt lines that the device uses to communicate with the software driver

running on the CPU. For example, the GPS receiver device "/soc/aips2/uart5/gps" depends on the UART bus it is attached to, while the UART controller depends on a 16kB addressable region of the AIPS peripheral bus (which connects to the system bus) and IRQ 30 of the interrupt controller. Hardware devices are bound to specific device driver implementations via the *compatible* property, which contains a list of string names that the software can use to match against available device drivers.

In addition to the information typically contained in the DT, the enforcement layer requires additional properties for the DT to serve as the trusted hardware specification. The *aio-deps* property contains the names of all non-standard properties that capture any dependencies on other devices; the set of standard properties includes *interrupts*, *interrupts-extended*, and *clocks*. The *aio-prot* property associates the device with a particular hardware protection domain; on the i.MX6 SoC, this would reference the CSU device with a resource specifier that contains the CSL register index that holds the access control configuration for the device. The *aio-class* property for each peripheral device to associate it with one or more high-level classes of devices (e.g., "Touchscreen", "Network").

Chapter 4: SeCloak: On/Off Control for Peripheral Devices

4.1 Introduction

In this chapter, I present my work on SeCloak. SeCloak enables users to reliably express on/off control over I/O peripherals on their device, such as radios (e.g., WiFi, Bluetooth) and sensors (e.g., GPS, microphone). For example, journalists interviewing confidential sources may want to use the microphone but reliably turn off the radios and GPS, and users may want to make sure the camera and microphone are disabled during private meetings. This guarantee holds even in the presence of malicious applications or compromise of the platform software, including the OS itself.

We[1] provide a simple abstraction to users of secure virtual switches that they can use to control the various peripherals on their device. We designed SeCloak to implement this abstraction, which acts as an independent, platform-agnostic enforcement layer beneath the existing platform and applications. In order to minimize

[1]This work involved collaborations with Rijurehkha Sen, Peter Druschel, and Bobby Bhattacharjee.

the trusted code base, we divided the design of SeCloak into two parts: an untrusted settings application and a trusted enforcement layer (the SeKernel). The user interacts with an untrusted settings application, which presents a traditional interface to the user. The application communicates the desired policy to the enforcement layer, which is responsible for requesting explicit user confirmation before applying and enforcing the settings based on a trusted specification of the hardware.

We implemented SeCloak by building on top of the ARM TrustZone hardware security extensions. The SeKernel runs as the secure world OS, which controls all memory and peripheral accesses of the non-secure world; the existing system software runs in the non-secure world. While TrustZone provides support for configuring access control to peripherals, there are several issues that make things more challenging in practice to satisfy the goals of SeCloak. First, it is common that the hardware provides access control settings at a granularity that is too coarse grained compared to the granularity at which the user wants to disable devices (i.e., not for individual peripheral devices). Either multiple peripheral devices may be grouped into a single protection domain, or they may share a common multiplexed bus which is the level that such access control is applied and enforced. Second, we need to be able to handle non-secure accesses to protected devices in such a way that maintains usability and stability of the system.

The primary contribution of this work is the design and implementation of an end-to-end system that satisfies all of our stated goals. The entire SeKernel, including secure device initialization, setting device state per user preference, user interaction for confirmation, and instruction emulation, is only 15k lines of code. (In

comparison, the Linux kernel is roughly 13m lines of code.) We believe our SeKernel TCB size satisfies our goal of having a minimal TCB, although more work could be done to reduce it further in practice (e.g., avoiding use of virtual memory). Finally, SeCloak requires 1 source line to be changed in the Linux kernel (the change can be applied directly to the kernel binary if the source is not available), introduces a new kernel module (for calls into the SeKernel), and no change whatsoever to other software layers (e.g., Android, applications).

The rest of this chapter is organized as follows. I present an overview of the design of SeCloak and how to map such a design onto ARM TrustZone in Section 4.3. I describe the design and implementation of the SeKernel in Section 4.4 and the non-secure software components in Section 4.5. I present the evaluation of the prototype implementation in Section 4.6, and finally conclude in Section 4.7.

4.2 Closely-Related Work

While I introduced related work in Chapter 2, I will take a deeper look at the prior work that is closely-related to SeCloak next.

Brasser et al. [80] enable on/off control of peripheral devices that are in restricted spaces (e.g., where the use of the camera is not allowed). The system relies on a local, TrustZone-isolated policy enforcement service that grants a remote policy server read/write access to system memory. To protect against rogue accesses, the user relies on a separate vetting server that determines whether to allow or deny each of the policy server's memory access requests. The policy server uses this re-

mote memory access capability to query the state of the platform OS and modify the OS's device configuration according to the desired policy. The policy server must be able to handle any platform OS version, configuration, and state, which increases its TCB and requisite maintenance over time. Also, the policy server cannot tolerate any vulnerability in the platform OS that is not known to the policy server (e.g., zero-day exploits), and must periodically re-check the state to ensure continued compliance. Like this work, SeCloak provides reliable on/off control of peripheral devices; however, it does so under a stronger threat model that includes a compromised platform OS. Additionally, SeCloak has a smaller TCB as it does not depend on any details of the platform OS in order to meet the requisite security properties.

Santos et al. [81,82] present "trust leases", which allow applications to request (with user approval) leases to place the device in a restricted mode until some terminal condition is met (e.g., after 4 hours). The trust lease model could be used to implement a settings application that allows the users on/off control over peripheral I/O devices. Their threat model assumes that the platform OS is trusted and correct; their prototype implementation lives inside the Android framework and Linux kernel. In contrast, SeCloak has a stronger threat model that includes a malicious platform OS, and operates as a separate, minimal secure kernel that runs alongside the existing platform OS.

PROTC [104] is a system for safeguarding flight control systems on drones from non-essential but malicious software. PROTC runs applications in the TrustZone non-secure world, and a kernel with access to protected peripherals in secure word.

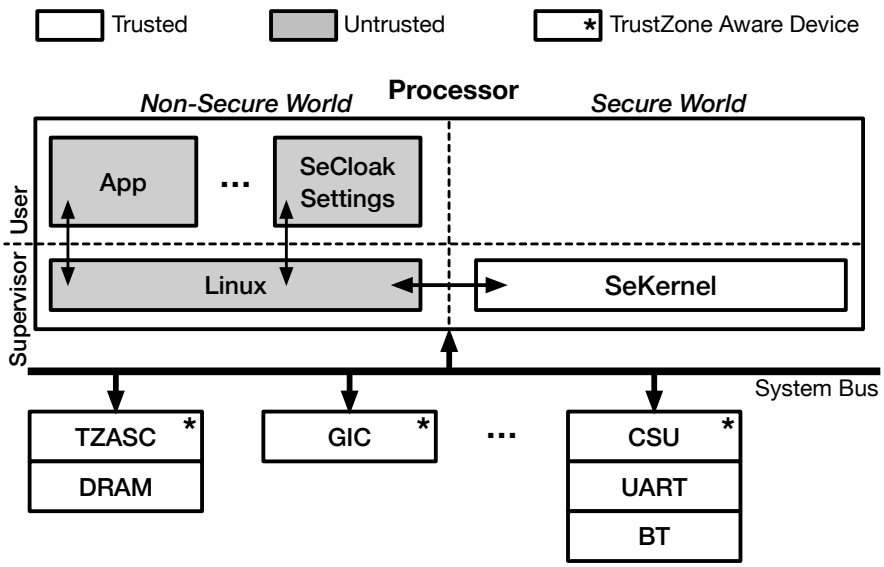


Figure 4.1: Overview of the SeCloak architecture mapped to an implementation on ARM TrustZone.

PROTC's secure kernel communicates with ground control using an encrypted channel. Compared to SeCloak, PROTC is designed for a different application domain, and does not allow for dynamic modification of protected peripherals. Untrusted applications are always isolated, and the secure kernel contains all of the protected device drivers.

Viola [84] enables custom, per-peripheral notifications whenever the I/O peripheral device is being used; for example, blinking the notification LED when the camera is active. Viola employs formal verification techniques to provide guarantees that the user will be notified. SeCloak and Viola are complementary: the user could use SeCloak to disable devices and rely on Viola to notify them when specific enabled devices are in use.

4.3 Design Overview

Figure 4.1 illustrates the SeCloak architecture that leverages underlying mechanisms provided by ARM TrustZone. In our design, Android/Linux and all applications run in the non-secure world, while our SeKernel enforcement layer runs in the secure world as both the monitor and the OS. The SeKernel is responsible for configuring the hardware protection mechanisms to isolate itself from other software on the system and to enforce the user's control policy settings.

As discussed in Section 4.2, many prior approaches have ported (parts of) device drivers into a secure kernel to provide robust access to peripherals, and for limiting the access given to the non-secure kernel. However, none of these systems satisfy the goals of SeCloak. Our work departs from these systems in two major ways: we do not modify the non-secure kernel— everything, including device drivers and interfaces, remain completely unchanged. Additionally, our secure kernel TCB is extremely small (see Section 4.6.1); we do not require a significant set of device drivers or complex mechanisms within the SeKernel to enable the functionality. In fact, the bulk of the functionality provided by the SeKernel relies on: 1) understanding the hardware configuration from a trusted hardware specification, and 2) trapping and emulating loads/stores to enable selective access to peripherals.

4.3.1 SeCloak Workflow

Figure 4.2 shows the SeCloak workflow. Users interact with a regular Android app (leftmost screenshot), and choose On or Off settings for various IO devices and

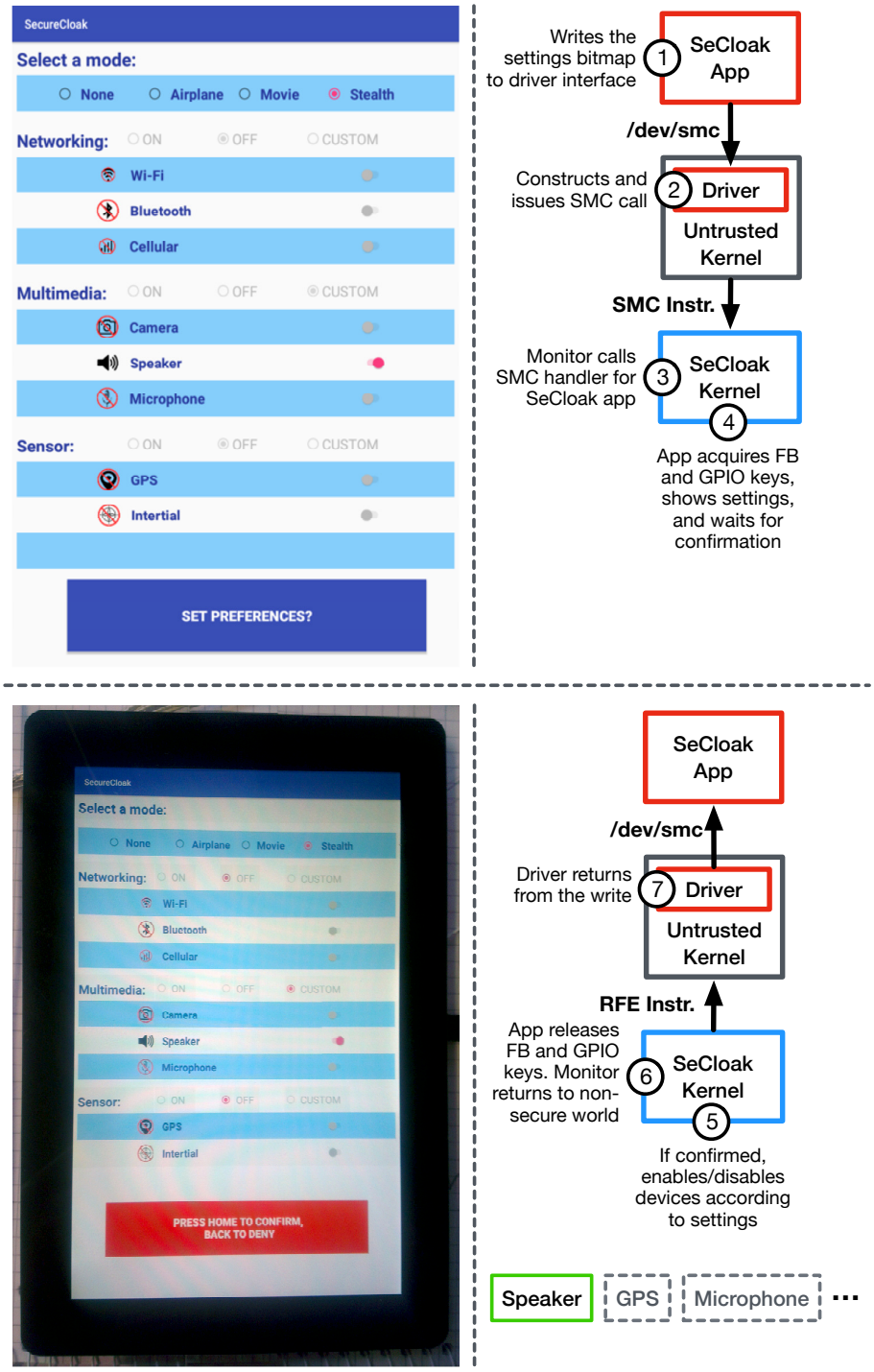


Figure 4.2: Overview of the SeCloak workflow. The first panel (leftmost) shows a screenshot of the non-secure Android app. The second panel shows the steps taken after the user pushes the button to apply the settings. The third panel shows a photograph of the secure SeCloak app, which (re-)displays the settings to the user and waits for user confirmation. (Screenshots are not possible in secure mode.) The fourth panel shows the steps taken if the user confirms the settings, such as disabling/enabling devices and returning control back to the non-secure world (and app).

peripherals. This app is similar to the "Settings" apps that are already available on smartphones and tablets.

An implicit assumption is that users understand which peripherals should be turned off (or on) under different circumstances. Towards this end, the SeCloak app helps users by providing pre-defined operating modes (e.g. Airplane mode or Stealth mode) and associated settings for the appropriate groups of peripherals. As we shall see next, SeCloak ensures that for any mode or sets of individual devices that the user chooses in this step, the user receives an unambiguous confirmation of their state.

Once the user chooses "Set Preferences?", the app uses a Linux device driver to invoke a cross-kernel call, called a `SMC` call in TrustZone (second panel from left in Figure 4.2). The argument to this call encodes the user's preferences. The SeKernel receives these preferences as part of the `SMC` call handler. Note that malicious software (the app, system services, the non-secure kernel) could have changed the preferences prior to the `SMC` call!

User preferences, possibly modified, are received within the SeKernel as part of the `SMC` call. At this point, the SeKernel takes exclusive access of the framebuffer and hardware buttons. The SeKernel parses the preferences, and recreates an image that exactly corresponds to the settings passed via the `SMC` call, and copies this images to the framebuffer. *If* the preferences had been modified, the image on the framebuffer would *not* correspond to the settings chosen by the user, and the user would notice a setting that does not correspond to their choice. The user is thus notified of malicious software on their device.

The SeKernel changes the "Set Preferences?" button to one that allows the user to confirm the settings by pressing a hard button (the "Home" button), or to go back (using the Back key) to the app and continue changing settings. We show a photograph of this screen (with a red secure confirm button) in the third panel of Figure 4.2.

A malicious app could display the preferences screen and then spoof the confirmation screen. It is imperative that during the confirmation phase, the user is unambiguously notified that she is interacting with the SeKernel. Thus, during this phase, the SeKernel lights a protected LED which ensures the user that she is interacting with the SeKernel. This LED is never accessible to the non-secure kernel.

Assuming the settings were as the user intended, she may confirm the settings by pressing the "Home" button. The SeKernel disables (or enables) various IO devices and peripherals as instructed (rightmost panel in Figure 4.2).

4.4 SeCloak Secure Kernel

We describe SeCloak's secure kernel, called the SeKernel, in this section. A signed device tree describes available hardware and protections to the SeKernel. Prior to describing the kernel itself, we discuss how it is securely booted (next), and the device tree.

Modern devices are equipped with secure, tamper-proof, non-volatile storage, into which device manufacturers embed (hashes of) public keys. The devices contain

a one-time programmable Boot ROM that has access to these keys, which are "fused" onto the hardware.

In bootstrapping SeCloak, we assume that a trusted principal (either the hardware manufacturer or the user) has performed the following steps:

- Generated a trusted key, and stored the key onto the tamper-proof non-volatile storage. For convenience, modern devices often allow multiple such keys to be "fused" onto the hardware; once installed, these cannot be removed or modified by software.
- Program the boot ROM to load a signed bootloader image. The boot ROM verifies the signature against the fused key(s) and then executes the bootloader if successful.
- The bootloader (`U-Boot` [105] in our case) contains a set of public keys which it uses to verify signatures on all loaded images. The bootloader will locate and load a signed SeKernel image and signed device tree blob (explained next). After verifying the signatures, the bootloader will execute the SeKernel. Modern bootloaders already support such verified booting.

4.4.1 Device Tree

The SeKernel needs to learn the complete set of available peripheral devices as well as the mapping between hardware protection mechanisms and devices. We use the device tree (DT), as described in Section 3.3, along with several additional properties such that it can serve as a trusted specification of the hardware for the

SeKernel. First, we add a "class" property that maps low-level components (such as interrupts and pins) to user-understandable names, such as "microphone", "WiFi", etc. These class strings correspond to individual devices that can be controlled via SeCloak. Second, we add a "protect" property that identifies hardware protection bits that must be set to protect the device. On our prototype, these map devices to their associated CSU registers.

4.4.2 SeCloak Kernel

Upon boot, the SeKernel initializes hardware defaults prior to launching the non-secure kernel. Specific steps include setting control and security registers to appropriate defaults, and setting memory protections such that the non-secure kernel cannot overwrite the SeKernel's state.

The SeKernel initializes its internal data structures by initializing the system MMU with virtual memory page table mappings for various regions, including regions for non-secure RAM, SeKernel heap, and for MMIO devices. The SeKernel also starts the non-boot CPUs, and initializes per-core threads and their contexts. Faults and calls from the non-secure kernel transition the CPU into a monitor mode, and the SeKernel initializes the secure monitor with its stack pointer and call vector. Finally, the SeKernel opens and parses the device tree.

4.4.2.1 SeKernel Device Drivers

The SeKernel itself contains minimal drivers for three devices: GIC (generic interrupt controller), GPIO controller, and the framebuffer. The GIC driver isolates interrupts that can be received by the non-secure kernel, the GPIO controller allows the SeKernel to directly interact with hardware buttons, and the framebuffer driver allows the SeKernel to render the confirmation screen. Together, they enable the secure part of SeCloak. We explain these drivers next.

GIC Driver The Generic Interrupt Controller (GIC) chip handles the distribution of interrupts to CPU cores and enables isolated control and handling of non-secure and secure interrupts. The SeKernel GIC driver supports functions to (1) enable or disable specific interrupts, (2) set CPU mask and interrupt priority, (3) assign interrupts to security groups, and (4) registering interrupt handlers. These functions allow isolating interrupts associated with specific devices to be either completely disabled or to be delivered to the SeKernel. The SeKernel can receive hardware interrupts and optionally re-deliver them to the non-secure kernel; for example, this functionality is used by the GPIO and GPIO keys drivers that we describe next.

GPIO Driver The general-purpose input-output (GPIO) controller supports input and output operations on individual hardware pins. In addition, the GPIO controller can also act as an interrupt controller on a per-pin basis. When an interrupt condition is triggered for a pin, the GPIO controller triggers a (chained) interrupt which is handled by the GIC.

The SeKernel GPIO driver supports acquiring/releasing pins for exclusive SeKernel use, registering an interrupt handler for a given pin, and reading (or writing) values from (or to) a pin. The GPIO driver relies on the GIC driver to register its own interrupt handler, which (when invoked) will read the GPIO device state in order to determine which pins raised the interrupt, and then invoke handlers corresponding to these pins. The driver protects and emulates accesses to the GPIO controllers in order to allow the non-secure world to continue to use any non-acquired pins while preventing it from inferring any information about the acquired (secured) pins.

Building on the GPIO driver, the GPIO keys driver supports hardware buttons/keys connected to GPIO pins (e.g., power and volume buttons). The GPIO keys handler translates hardware button presses into a key code that is specified in the device tree (e.g., `KEY_BACK`) and passes it on to any SeKernel listeners. The listeners can choose to consume the key press or allow it to be passed back to the non-secure world. We use the GPIO keys driver to register listeners for a secure shutdown sequence (see Section 4.5.2), and also for the cloak application to wait for the user to confirm or deny the displayed settings.

Framebuffer Driver The SeKernel framebuffer driver uses the image processing unit (IPU) device to display images. When the SeKernel application acquires the framebuffer, the driver allocates a single buffer in the secure region of memory and sets the buffer format (RGB24) in the IPU. Additionally, the driver protects access to the IPU and emulates accesses in order to prevent the non-secure world from

overwriting the settings (see Section 4.4.6.1 for emulation policy details). When the SeKernel application releases the framebuffer, the driver restores the previous settings of the non-secure world and unprotects the IPU. Additionally, the framebuffer driver provides helper functions for clearing the buffer with a single color and for blitting images onto the display at specified locations. We rely on the framebuffer driver for (re-)displaying the settings in the SeCloak app. The images displayed by the SeKernel framebuffer driver cannot be modified by the non-secure kernel.

4.4.3 `SMC` Handlers

The SeKernel supports two `SMC` calls, CLOAK_SET and CLOAK_GET, from the non-secure kernel to enable SeCloak.

The non-secure kernel invokes the CLOAK_SET call with a bitvector as the argument. Individual bits in the bitvector correspond to the settings for different device classes. The bitvector contains "special" bits that encode modes (e.g., Airplane, Movie, Stealth), and groups (e.g., Networking) as displayed by the app.

The CLOAK_SET handler executes the following steps:

- It starts by acquiring the framebuffer and GPIO keypad (via GPIO). As described in Section 4.4.2.1, acquiring devices applies necessary hardware protection settings, emulation policy, and initial settings for the secure use of the device.

- The CLOAK_SET handler parses the bitvector and checks to see if it is valid; if so, it uses framebuffer driver routines to blit corresponding images to the

screen in order to (re-)display the settings to the user.

- Next, the notification LED, which is persistently acquired for exclusive use by the SeKernel, is turned on by the handler (via GPIO) to notify the user that the SeKernel is in control.

- The CLOAK_SET handler then waits for the user to confirm (via the 'Home' button) or deny (via the 'Back' button) the settings via its registered GPIO keypad listener. If the user confirms the settings, the handler will issue calls to enable or disable each device class; otherwise, if the user denies the settings, the handler does not take any action.

- Finally, the handler releases the acquired devices (which resets per-device state as necessary, e.g., framebuffer formatting and addresses) and returns.

In order to disable (or re-enable) a device class, the CLOAK_SET handler first identifies all devices that belong to the given class (as described in the device tree). For each of those devices, the handler locates any "protect" properties, which identify the hardware protection that must be set to isolate the device. In some cases, the device itself may not have hardware protection, but the bus it is located on may. Thus, the code must search for possible hardware isolation not just at the device node, but recursively up the device tree as well. In this way, the SeKernel applies the hardware isolation for each device as described in the device tree.

The non-secure kernel can use the CLOAK_GET call to receive a bitvector that encodes the current protection state of device classes (and which mode is active or

which groups are enabled or disabled). Upon launch, the non-secure Android app uses this call to render an initial setting.

4.4.4 Non-Secure and Secure Device Sharing

We rely on the ability to share devices between the non-secure and secure worlds, such as for providing a secure shutdown sequence via the GPIO keypad (e.g., power and volume buttons) while still allowing the non-secure world to handle button presses. Two underlying mechanisms enable such sharing: 1) redelivering interrupts to the non-secure world, and 2) emulation policy to control the non-secure view of the device. While explaining the mechanisms, we will focus on the example of the GPIO controller. The GPIO controller uses a GIC interrupt in order to signal that the interrupt condition is met for one (or more) pins.

In order to share interrupts with the non-secure world, we modify the device tree such that the SeKernel operates on the actual hardware interrupt line of the device that is connected to the GIC, while the non-secure kernel operates on a (previously unused) interrupt line. When the SeKernel receives an interrupt that should be shared, it sets the corresponding non-secure world interrupt line pending via the GIC.

As part of handling the GPIO interrupt, the driver must acknowledge the interrupt using the Interrupt Status Register (ISR). The ISR contains a bit for each pin that is 1 if an interrupt condition is met; writes to the ISR acknowledge the interrupt and reset the corresponding bit to 0. The SeKernel specifies an emulation

policy on the region of memory associated with the GPIO device, such that it can control the reads and writes to the various device registers (including the ISR). When the SeKernel shares an interrupt with the non-secure kernel, it keeps track of this in a bitmask. Later, when the non-secure kernel reads from the ISR after receiving an interrupt, the SeKernel policy returns the value of the ISR OR'd with the sharing bitmask (which is needed to handle edge-triggered interrupts). After handling the GPIO interrupt (e.g., invoking the GPIO keypad handler), the non-secure kernel writes a 1 to the pin to acknowledge and clear the interrupt; the SeKernel applies policy to this write by clearing the bit in the sharing bitmask. Additionally, this policy extends to support SeKernel-only pins by maintaining a secure mask which is used to modify the writes and reads made by the non-secure kernel (e.g., masking the pins in the ISR corresponding to SeKernel-only pins).

4.4.5 DMA

Devices that are DMA masters can issue memory accesses on the system bus. For example, the Image Processing Unit (IPU) will perform periodic DMA transfers to read from framebuffers (whose addresses are specified in the IPU's registers). Each DMA master has permissions assigned for its bus accesses (i.e., non-secure or secure), which (on our platform) are configured in the CSU registers. In order to prevent the DMA masters from reading (or even modifying) the SeKernel memory, we must configure their accesses as non-secure. However, this presents a problem for the IPU device: since we need to present a secure framebuffer to the screen, it

must be able to perform DMA accesses to secure memory regions. To address this, we use the TZASC to configure the region that contains the framebuffer as non-secure read and secure read/write. While this lets the non-secure kernel inspect the secure framebuffer, we do not require confidentiality of this framebuffer for any of our security goals (only the integrity of its contents).

4.4.6 Instruction Faults and Emulation

The SeKernel configures TrustZone such that accesses by the non-secure kernel to memory regions that belong to protected devices cause a fault. This fault is trapped by the monitor mode handler of the SeKernel. We need these traps to be able to selectively allow or deny non-secure kernel accesses to devices.

For a rudimentary SeCloak app, it is sufficient to simply configure TrustZone protections, and ignore these faults. However, such a solution is unworkable if we want the device to remain usable, as per our original goals, when specific peripherals are protected. In general, the SeKernel has to trap the faulting instructions, and selectively emulate them based on hardware state as we describe in this section.

There are two main reasons to intercept non-secure accesses to protected resources and emulate these accesses in the secure world. First, there can be a mismatch between the granularity of hardware protection and that of individual devices that are being protected. For instance, on the i.MX 6 [106], the Central Security Unit (CSU) contains Config Security Level (CSL) registers that restrict access to peripheral devices according to whether the accesses are made by the non-secure or

secure world. These CSL registers group multiple devices into a single register (e.g., `GPIO1` & `GPIO2` or `PWM1` through `PWM4`). If we want to disable a single (or subset of) devices, we must allow accesses to all others that are protected by the CSL group. Dependencies in the device tree can also cause mismatches in hardware–software protection granularity. For example, the ft5x06_ts touchscreen uses a GPIO pin to signal an interrupt to the processor when the user is touching the screen; in order to secure the touchscreen, we must also secure the *individual* GPIO pin, but not all the 64 pins that are protected by the corresponding CSL register. Additionally, there may be a single bus device with hardware protection capability that multiplexes communication to multiple slave devices. For instance, an I2C bus may serve both the touchscreen and camera devices; however, if the user disables the camera device, the touchscreen should still be usable.

Second, we can use emulation for efficiently acquiring devices for (temporary) exclusive use by secure applications, as well as to share devices between the secure and non-secure worlds. This can reduce the trusted codebase in the SeKernel, e.g., by allowing non-secure kernel writes to the device for non-critical accesses. We use this technique to reduce the driver code size for the framebuffer driver.

4.4.6.1 Instruction Emulation: Detail

Each access to a device ultimately performs a memory-mapped Input/Output (MMIO) read or write operation to a region of memory associated with the device. (The mapping of memory region to device is obtained from the device tree.) When

hardware protections are enabled for a particular device, MMIO accesses produce data abort exceptions; these are traditionally handled by the non-secure kernel.

Hardware setup In order to intercept these accesses, SeKernel sets up the Secure Configuration Register (SCR) in the CPU to specify that all external aborts should be handled by the monitor. This setting causes data aborts to signal a fault that transitions the CPU into the secure monitor mode. The faulting address and related information is available to the monitor fault handler.

In the SeKernel, the secure monitor fault handler invokes a routine that determines whether to emulate or deny the access and, if emulated, whether to modify the value being read or written. The SeKernel maintains a data structure that contains regions of memory (physical base address and size) corresponding to different devices, along with the prevailing policy for each.

The policy associated with each region may choose to deny a read or write. If a read is allowed, the value that is read can be modified prior to being returned to the non-secure kernel. If a write is allowed, the value to be written can be modified prior to the write. The SeKernel code decodes the instruction that cause the original fault: these instructions are of the form `ldr` (load register for reads) or `str` (store register for writes). By default, the SeKernel emulates the instruction exactly and returns control back to the non-secure kernel.

NS-kernel Execution Figure 4.3 shows this process. Here a non-secure driver attempts to read from a device ("Dev 1") that is disabled by SeCloak. Specifically,

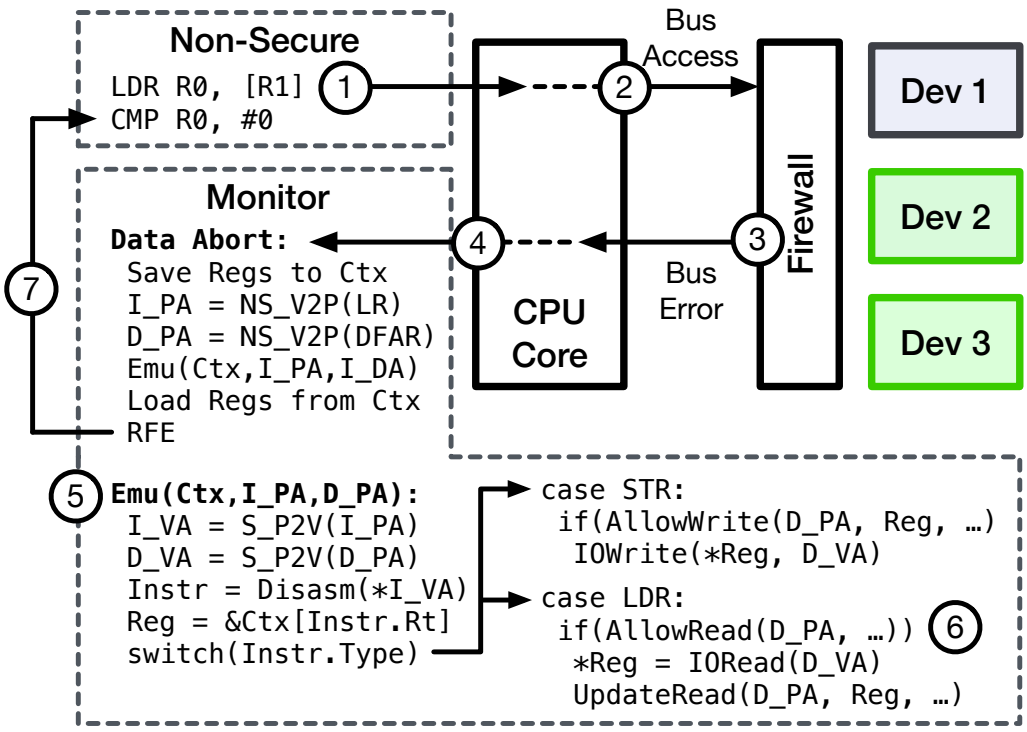


Figure 4.3: Components and steps involved in intercepting and emulating accesses made by the non-secure world.

the non-secure kernel code issues a `LDR R0, [R1]` instruction to load from the address pointed to by `R1` into `R0`. The `R1` register contains a memory address that belongs to "Dev 1".

Upon executing this instruction (1), the CPU issues a read on the system bus (2). This memory read is intercepted by the hardware firewall (CSU/TZASC) responsible for protecting "Dev 1". The firewall checks to see if the access should be allowed; if not, the firewall returns a bus error to the CPU (which interprets this as an external data abort). In this case, since "Dev 1" is disabled by SeCloak, the firewall will deny the read and report a bus error to the CPU (3).[2] The CPU receives this bus error which corresponds to an external data abort. Given the SCR

[2]The system can be configured to also issue an interrupt to the CPU upon such an error; we do not use this option in our implementation.

configuration, the CPU switches to monitor mode and invokes the monitor mode's data abort handler (4).

The data abort handler saves (and later restores) the current register set [3] and preserves the location to return to (i.e., the instruction *following* the faulting instruction).

Fault Handling At this point, the fault handler has to determine two items: what was the instruction that caused the fault, and what was the faulting address? By convention, the Data Fault Address Register (DFAR) contains the virtual address of the access that caused the fault, and the LR register contains the virtual address of the instruction that caused the fault. However, these virtual addresses are *non-secure* virtual addresses, and the fault handler uses an ARM co-processor routine to resolve them into physical addresses ("D_PA" from the DFAR, and "I_PA" from the LR). The fault handler then passes control to the SeKernel emulation routine with the saved register context of the non-secure world (called "Ctx") and these two addresses as arguments.

The emulation routine (5) begins by translating these physical addresses to secure-world virtual addresses, and checks to make sure that they are in appropriate regions: the non-secure RAM for the instruction and a device MMIO region for the data address. Next, the emulation routine invokes a custom instruction decoder we have written to decode the instruction and determine the type of instruction

[3] To be precise, some registers in ARM are "banked" (e.g., the link register `LR`), in that each mode has its own copy of the register. The abort handler saves the non-banked registers as well as the `LR` corresponding to the mode that caused the abort.

(whether it is a load or store) and the register involved in the transfer.

Once the instruction and the physical address is decoded, the prevailing policy (e.g., deny or allow with modifications) is implemented as described above. In this case, the access is made by a load instruction, so the emulation first checks to see if the policy allows the read, performs the IO read operation, and finally checks to see if the policy wants to modify the result (6). If allowed, the final result is stored in the NS-context structure (that contains the non-secure registers).

In order to handle the case where multiple slave devices share a bus, a bus-specific policy must be provided. The device tree contains resource information that specifies how each device will be addressed on the bus; for instance, in the case of I^2C, this corresponds to the 7-bit slave address assigned to each device. The policy operates over the accesses to the bus's MMIO region to determine which device is being accessed, such that it can deny accesses to disabled devices (while allowing all others).

NS-kernel resume The emulation routine then returns control back to the data abort handler, which restores the registers for the non-secure world from the NS-context data structure. Note that for reads, one of these registers may now be updated as a result. Once the data abort handler terminates, the non-secure kernel continues by executing the instruction directly after the one that caused the data abort (7).

4.5 Non-Secure Kernel

Figure 4.2 shows a screenshot of the SeCloak Android app in the left-most panel. The current version of the app is simple, allowing users to set ON/OFF preferences for the devices on our prototype board. Along with individual devices, the app allows users to choose different operating modes (e.g., Airplane, Stealth) and also provides the state of groups of peripherals (e.g., all networking devices.)

Once the user presses the "Set Preferences?" button, the app invokes a JNI call with a bitvector that encodes the user preferences. The JNI module uses a Linux `ioctl` call to pass the bitvector to the SeCloak kernel module which, in turn, issues the `SMC` call to the SeKernel with the bitvector as an argument.

4.5.1 A modification to the non-secure kernel

Recall that our design goals were not to modify the non-secure kernel or existing software if at all possible. Unfortunately, without a single byte modification, as we describe next, we can only provide the security guarantee, but not maintain system stability.

SeCloak requires that the SeKernel be able to trap individual accesses to protected devices and selectively emulate these instructions. However, as normally compiled, a Linux binary on ARM does not raise data aborts that identify the *specific* instruction that cause the abort. This is because ARM supports the notion of precise or imprecise data aborts, which reflects the ability (or inability, respectively) for the data abort handler to determine the exact instruction that caused the abort

from the value of the LR register. For precise data aborts, LR points directly (modulo a fixed offset) to the instruction that caused the abort; however, for imprecise data aborts, it points to an arbitrary instruction that was executing at that time the abort was raised.

Whether an instruction raises a precise or imprecise abort depends on the page table entry (PTE) attributes of the memory that the instruction attempts to access. Precise data aborts are triggered for ldr and str instructions that access "strongly-ordered memory". Strongly ordered memory does not allow accesses to be buffered by either the processor or bus interconnect [107]. However, by default, memory regions associated with devices are mapped (e.g., via ioremap in Linux) with the DEVICE_NONSHARED attribute which does not impose strong ordering on the accesses (unlike the DEVICE_SHARED attribute, which does). With this default configuration, the SeKernel's data abort handler receives imprecise aborts and, while the handler can determine the faulting address via the Data Fault Address Register (DFAR), it cannot accurately determine the address of the instruction that caused the abort from the link register (LR). As a result, we must modify Linux such that it configures its device memory mappings to raise precise aborts, using the DEVICE_SHARED PTE attribute. While we do this step directly in the source, it is a simple change that can, in fact, be applied on the binary kernel image itself.

In our design, the SeKernel assumes that the non-secure kernel is "compliant" in setting device memory to be strongly-ordered. However, a non-compliant non-secure kernel can still not access protected devices. It will, however, likely not receive any useful service from protected device groups due to faulty emulation.

Kernel Module The SeCloak app requires a kernel module to invoke `SMC` calls, and we have added such a module to Linux. (Later versions of the Linux/ARM kernels already provide a standard `SMC` interface like our kernel module does, though even these kernels would require a module to export a userspace interface.) The kernel module provides a `ioctl` interface, which is used to communicate the user-selected bitvector to the non-secure kernel.

Framework Calls Along with the single change to the non-secure kernel, the SeCloak app also issues Android calls to address application and system stability. Note that these are not *changes* to the framework, but instead, extra calls that are invoked by the SeCloak app.

When the user elects to disable certain devices, the SeKernel configures hardware firewall mechanisms to prevent all accesses to the disabled devices. If the hardware device is attached to the system bus, then MMIO writes are discarded and reads return 0; otherwise, if attached to a peripheral bus, then bus access functions will return an error. Ultimately, device drivers are responsible for handling these errors, which typically involve several retries before abort. These errors will further propagate to system services (and applications) that are attempting to use the device; for example, when the camera is disabled and the user attempts to run the camera app, an error message appears after a few seconds.

Within the kernel, power management (PM) routines in device drivers rely on the ability to communicate with their devices in order to save relevant state and direct it to enter a low-power mode. When a device is disabled by the SeKernel,

these PM routines will fail and thus keep the device in a high-power active mode. In adverse cases, the inability to transition individual devices into low power states can prevent the entire system from being able to transition to a low-power state, such as suspend-to-RAM. Second, some device drivers may not contain appropriate error handling to gracefully recover from errors resulting from the denied accesses.

Therefore, the SeCloak app makes use of available system services (e.g., WifiManager with `setWifiEnabled`) to disable devices prior to configuring the hardware firewall mechanisms (and likewise enable devices after removing the hardware firewall restrictions). We use these system services for both the WiFi and Bluetooth devices. Once again, these untrusted system services need not be used (or trusted) in order to meet the specified security goals, but improve system availability and usability.

4.5.2 Device Reset

After a peripheral has been secured, malicious software inside the non-secure kernel or framework can try to subvert security by rebooting the entire device. Such a reboot could happen without the user necessarily noticing (while the device was idle) and could even be remotely triggered.

One option is to make device policies persistent in the SeKernel, such that they would be applied whenever the device is booted. While technically feasible (and indeed quite trivial), this option affects usability. Upon boot, the non-secure kernel (Linux) probes available devices based on the device tree, and may not set up

the device files and other software correctly if the probe fails (which it would if the device were secured upon boot.) In turn, parts of the Android framework may not initialize, leaving the device in a unstable/unusable state. Without a kernel rewrite or support, it is difficult (if not impossible) to uniformly re-enable devices that were protected at boot.

Instead, we adopt the following policy: the non-secure kernel can reset the device only if there are no disabled devices. Otherwise, the SeKernel does not allow the non-secure kernel to invoke PSCI [108] calls that are used to reset the processor.

This design choice has the following implications: first, no code, including remote exploits, in the non-secure kernel can reboot the device if any peripheral is protected. On the other hand, when the device is rebooted, the regular non-secure kernel probes can proceed as usual, and the device reboots in a fully usable state. Further, the SeKernel does not need to keep persistent state about policies, since the device always reboots with all peripherals accessible to the non-secure kernel. When the non-secure kernel needs to reset the device (e.g., after OS or software updates), the user must first run the SeCloak app and remove all protections.

The user may, at times, need to reboot the device after protection has been applied. For instance, the non-secure kernel or Android may become unresponsive due to bugs or attacks. To address this scenario, within the SeKernel, we recognize a hardware key sequence that the user can input to initiate a reset. Since physical user input is necessary for the device to be reset, this is a safe option, in that the user is aware that the device is booting into a unprotected state.

4.6 Evaluation

We use the Boundary Devices Nitrogen6Q development board to run our experiments, which contains an i.MX6 SoC with a quad-core ARM A9 processor with TrustZone security extensions. We use Android Nougat 7.1.1 with the Linux kernel version 4.1.15, both of which are provided by Boundary Devices. The SeKernel implementation is based on our custom fork of OP-TEE [60]. OP-TEE is a OS for implementing secure applications over TrustZone; SeKernel heavily modifies and reduces the OP-TEE codebase. Specifically, SeKernel retains OP-TEEs kernel threading and debugging support. SeKernel's MMU code is also based on OP-TEE. The device drivers required for SeCloak (e.g., framebuffer and GPIO keypad), device tree parsing, instruction interception and emulation, and the code for securing device state was developed specifically for the SeKernel.

We first present results to quantify the size of the TCB, both in terms of the lines of code as well as the interface exposed to the non-secure kernel. Next, we evaluate the overhead due to intercepting and emulating accesses and show that, while there is a fair amount of overhead for individual instructions, the reduction in overall system performance is negligible.

4.6.1 Size of TCB

In Table 4.1, we show a breakdown of the lines of code for our SeKernel implementation. "Core" consists of all non-driver and non-library code in the SeKernel. This code handles core SeKernel functionality, such as: memory man-

LOC Breakdown					
Type	**C Src**	**C Hdr**	**ASM**	**Total**	**Stmt**
Core	3233	2357	1391	6981	3781
Drivers					
CSU	45	9	0	54	29
Device Tree	401	57	0	458	261
Frame Buffer	146	29	0	175	113
GPIO	562	15	0	577	284
GPIO Keypad	169	14	0	183	89
<Other>	579	167	0	746	265
Drivers Total	1902	291	0	2193	1041
Libraries					
libfdt	1220	350	0	1570	840
bget/malloc	1421	68	0	1489	797
<Other>	1479	1182	81	2742	1212
Libraries Total	4120	1600	81	5801	2849
Total	9255	4248	1472	14975	7671

Table 4.1: Breakdown of the lines of code (LOC) for different parts of our SeKernel implementation. We list the LOC according to the language used (and source versus header) along with the total LOC. "Stmt" refers to number of statements, which counts lines in assembly (ASM) and semi-colons in C source and headers.

agement, threading, the secure monitor, SMC handling (e.g., PSCI and CLOAK). "Drivers" consists of all driver code, which is further broken down into specific drivers that we added to OP-TEE. The "<Other>" category contains pre-existing drivers, such as the UART (i.e., console), GIC, and TZASC-380 drivers. The "Frame Buffer", "GPIO", and "GPIO Keypad" drivers are smaller than their Linux counterparts since the secure drivers do not need to support all device functionality.

As listed under "Libraries", our device tree parsing code relies on libfdt to extract information from the flattened device tree file that the bootloader places into RAM. Additionally, the SeKernel uses the bget and malloc support for dynamic memory allocation. Finally, there are several other libraries and sets of functions that are aggregated as "<Other>", such as: snprintk and trace functions (for printing debug info), qsort (for sorting memory regions data structures), and common standard library functions (e.g., memcpy, strcmp). In general, we could further reduce our reliance on these libraries but leave this for future work.

In total, our SeKernel comes to just under 15k LOC (~7.7k statements). The SeKernel has a limited attack surface in terms of the interfaces that the SeKernel provides to the non-secure kernel, namely CLOAK_SET and CLOAK_GET. CLOAK_SET takes one argument, which is a bit vector containing the modes, groups, and classes that the user wishes to enable or disable; CLOAK_GET takes no arguments.

	Instruction Time (μs)	
Execution	**Load (`ldr`)**	**Store (`str`)**
Linux	0.11	0.29
Linux+SOM	0.27	0.33
Emulated	1.14	1.19

Table 4.2: Time to execute ARM instructions in the non-secure world that make device accesses. "Linux" execution uses the baseline Linux kernel without any changes. "Linux+SOM" execution uses the baseline Linux kernel but changing the device memory regions to enforce strong ordering of accesses. For "Emulated" execution, we configure the SeKernel to protect access to the WiFi controller and emulate the instructions that result in data aborts.

4.6.2 Emulation Overhead

We perform two experiments to analyze the performance overhead introduced by emulating non-secure instructions that access devices. We focus on the case where the emulation is allowed, such as when devices are shared between the non-secure and secure world (e.g., GPIO) or when multiple devices (one of which is disabled) belong to the same hardware firewall protection group.

In Table 4.2, we show the time taken to execute a single ARM load (`ldr`) or store (`str`) instruction that access a 32-bit device register on the WiFi controller. We issued each instruction one million times to compute the time taken for each individual instruction, and averaged this time over five trials. We varied the execution between "Linux", "Linux+SOM", and "Emulated" modes. For "Linux" execution, we use the baseline Linux kernel without any changes, while for "Linux+SOM" we change the attributes for device memory regions to enforce strong ordering of memory accesses (required for interception and emulation, see Section 4.4.6). For "Emulated" execution, we configure the SeKernel to protect access to the WiFi con-

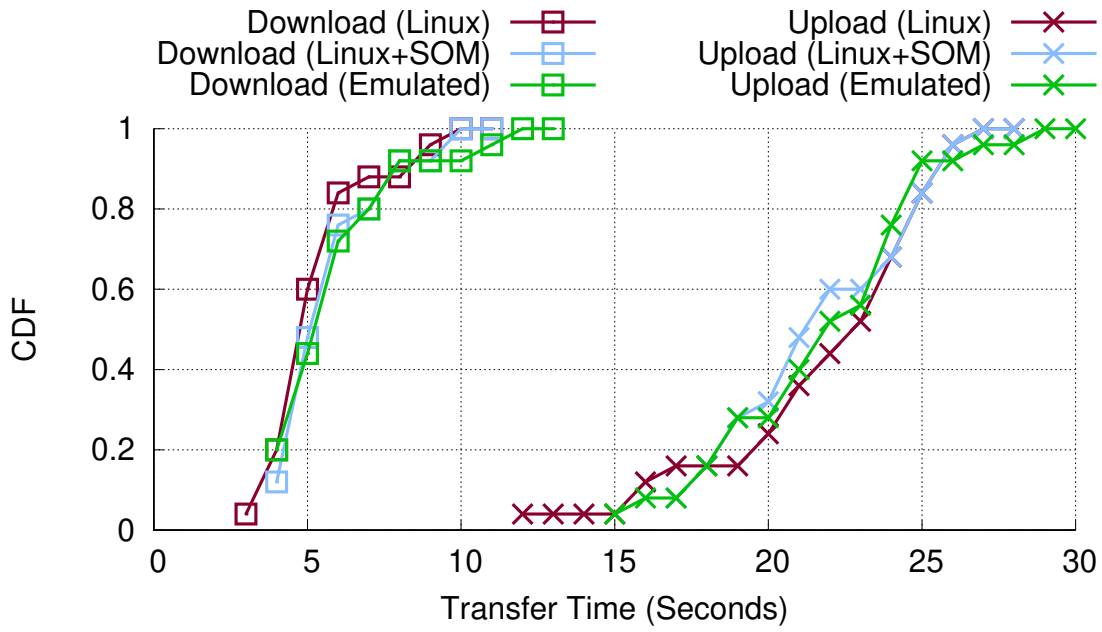


Figure 4.4: Time taken for upload and download transfers of a 10 MB file to complete over WiFi. "Linux", "Linux+SOM", and "Emulated" correspond execution modes evaluated in Table 4.2

troller via the hardware firewall (i.e., CSU) registers and set the emulation policy to allow accesses to the WiFi controller's MMIO registers.

The requirement of strongly-ordered memory accesses imposes overhead as expected, as seen by comparing "Linux" and "Linux+SOM"; there is a increase in instruction time of 2.45x and 1.14x for loads and stores respectively. We also see a further increase in instruction time from "Linux+SOM" to "Emulated" of 4.22x and 3.61x for loads and stores respectively. Note that, even though tapping and emulating accesses incurs a fair amount of overhead, high-throughput devices (e.g., camera, network, and display) rely heavily on DMA transfers for performance and should remain largely unaffected by emulation overhead (which only affects the control path for DMA). To that end, we next take a look at a macro-level benchmark involving the WiFi controller.

Figure 4.4 show the time taken to transfer files over WiFi when the controller accesses are emulated (or not). We used the WiFi Speed Test [109] application to perform the experiments and log the time taken for each of the trials; we used a laptop as the other endpoint for the file transfers.

The x-axis shows the time taken by the transfer in seconds, and the y-axis shows the cumulative fraction of transfers that completed within a given time. Each CDF in Figure 4.4 is computed over 25 runs.

The download and upload performance shows that there is no visible impact of interception and emulation on WiFi transfers, despite an appreciable increase in execution time for individual load and store instructions (as shown in Table 4.2). This is because the WiFi driver and controller, like all modern bulk data transfer devices, uses DMA to transfer packets. Once the controller firmware is loaded, and the DMA tables configured, each packet transfer (which can be many thousand bytes) requires very few (tens) MMIO instructions to initiate the DMA. We believe this result indicates that SeCloak can be used, even for high performance peripherals, without significant impact on user-perceived performance.

4.7 Conclusion

In this chapter, I have describe SeCloak, a system which employs a small-TCB kernel to allow users to unambiguously and verifiably control peripherals on their mobile devices. Such a capability has many uses, e.g., it can allow users to ensure they are not being recorded, or journalists to ensure that they are not being

tracked by using the WiFi/Bluetooth radios (or other means). The main technical challenge in designing SeCloak was to ensure that existing mobile device software, in particular Android and Linux, could co-exist with the secure kernel without code modification and without affecting device stability and usability. Towards this end, we have described an instruction emulation mechanism that enables SeCloak without changing existing software using a very small secure kernel.

Chapter 5: AIO: Control and Assurance Over Sensitive I/O Data

5.1 Introduction

In this chapter, I present my work on AIO [2]. AIO is an enforcement layer that enables transparency and expressive, end-to-end control over the software that collects, processes, and shares I/O data. AIO provides a framework that allows users to directly specify and reason about high level data disposition policies, e.g., "my touchscreen PIN input should only be available to the bank server" or "no untrusted application may see data from the GPS sensor". AIO ensures that these policies are enforced regardless of the behavior or compromise of platform software, including the OS. AIO relies on an ability to *isolate* itself from other software on the device, and to *mediate* interactions between software and hardware resources. These properties are derived from key enabling mechanisms provided by the hardware layer, in the form of virtualization and security extensions.

AIO inherently recognizes and supports many mutually distrusting principals (e.g., device manufacturer, user), providing a means for: 1) scoping trust to each

principal, 2) safely composing software from multiple principals, and 3) enabling more-privileged principals enforce policies over paths constructed by less-privileged principals. Policies can be used to construct and compose various forms of assurance over I/O data, such as confidentiality and provenance; AIO provides a way to prove these assurances through (remote) attestation.

The AIO enforcement layer is a software component that provides necessary mediation functionality, but the interface it provides is too low-level for direct use by users or even application developers. We[1] introduce a new abstraction, an "accountable path", that can be used to easily specify and constrain data flow within a device. Accountable paths can be used to program AIO to provide useful and easy to reason about properties on how sensitive data is collected and processed within a AIO-capable device.

The contributions are as follows:

- We introduce the accountable path abstraction, which resolves an impedance mismatch between the assurances and control desired by users (and other principals) and the mechanisms supported by existing platforms. This abstraction also provides a useful way to reason about the state of the system, through attestations over (parts of) paths.
- We provide a design of the AIO enforcement layer, which executes as a new, isolated, lowest-layer of software and implements the accountable path abstraction. Our design is focused on limiting the trusted computing base (TCB) and attack surface of this critical system component.

[1]This work involved collaborations with Peter Druschel and Bobby Bhattacharjee.

- We implement and evaluate a prototype of our AIO enforcement layer, leveraging ARM TrustZone hardware security extensions to meet our isolation and mediation requirements. This prototype is targeted at the Nitrogen 6X development board, which has hardware that matches most contemporary smart devices. In our implementation, we augment Android/Linux with support for accountable paths.
- We demonstrate the functionality and utility of accountable paths in the context of several application scenarios, such as a banking application with secure transaction confirmation. We also demonstrate the practicality of our approach through a suite of benchmarks.

The rest of this chapter is organized as follows. In Sections 5.2 and 5.3, I present our accountable path abstraction and describe the design for AIO. In Section 5.4, I discuss a prototype implementation of AIO on top of ARM TrustZone. In Section 5.5, I introduce a toolkit that contains many building blocks for developing end-to-end secure applications with accountable paths, and discuss how these are used in various application scenarios. In Section 5.6, I evaluate the performance of these applications running on top of AIO, as well as microbenchmarks for AIO itself. Finally, I conclude in Section 5.7.

5.2 High-level Overview

Figure 5.1 shows a high-level overview of the AIO architecture. AIO runs as an isolated, privileged layer within the system in order to: 1) mediate accesses to hardware devices, 2) mediate communication between isolated software modules. AIO

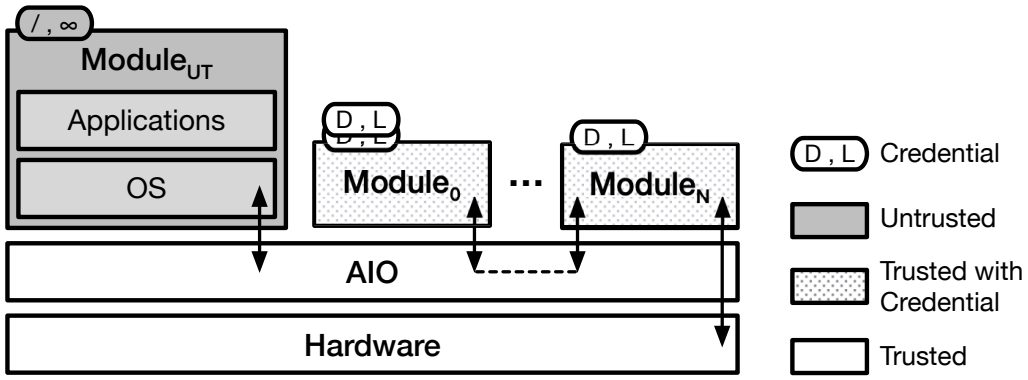


Figure 5.1: A high-level overview of the AIO architecture.

treats the existing platform software, such as the OS and applications, as a single, untrusted module (Module$_{UT}$). AIO executes each module in an isolated (trusted) execution environment, mediating the module's access to underlying hardware devices as well as to other modules. This mediation enables enforcement of policies on inter-module communication and limits the memory/device accesses made by modules.

AIO recognizes mutually distrusting principals, including the device manufacturer, platform provider, user (device owner), and various application and policy principals. By default, the user is given full privileges on their device, and may delegate these privileges to other principals, by providing them with (revocable) *credentials*. A credential represents a capability assigned to a principal; a capability provides access rights to a set of hardware device resources and a privilege level for these accesses. In turn, principals may assign these credentials to the modules they load. Given a credential, a principal may derive and delegate a (less-privileged) credential to another principal that has a reduced scope of hardware device resources and/or a lower privilege level.

Next, we introduce accountable paths, and discuss how they map to the AIO module structure, followed by an end-to-end example that shows how accountable paths can be used to build useful applications.

5.2.1 Accountable Paths

At a high-level, an accountable path represents a software path between hardware resources and a software endpoint; the endpoint may be an application running on the local device or a (trusted) remote server. Accountable paths are composed of a sequence of modules that are bound at runtime by AIO. Accountable paths do not impose a-priori rules on the software composition of modules; as such, they can be device drivers, critical components of applications, or software layers that implement well-known interfaces.

Accountable paths support safe composition of software from different principals, such that more-privileged modules may extend or replace functionality implemented by less-privileged modules. Accountable paths also enables more-privileged principals to control paths constructed by less-privileged principals by applying *policies* along the path. Next, we walk through an end-to-end example to illustrate how these ideas can be used to create useful applications using accountable paths in AIO.

5.2.2 Workflow Overview

Figure 5.2, shows a high-level overview of an exemplary workflow that uses accountable paths and AIO to build an end-to-end secure mobile banking application.

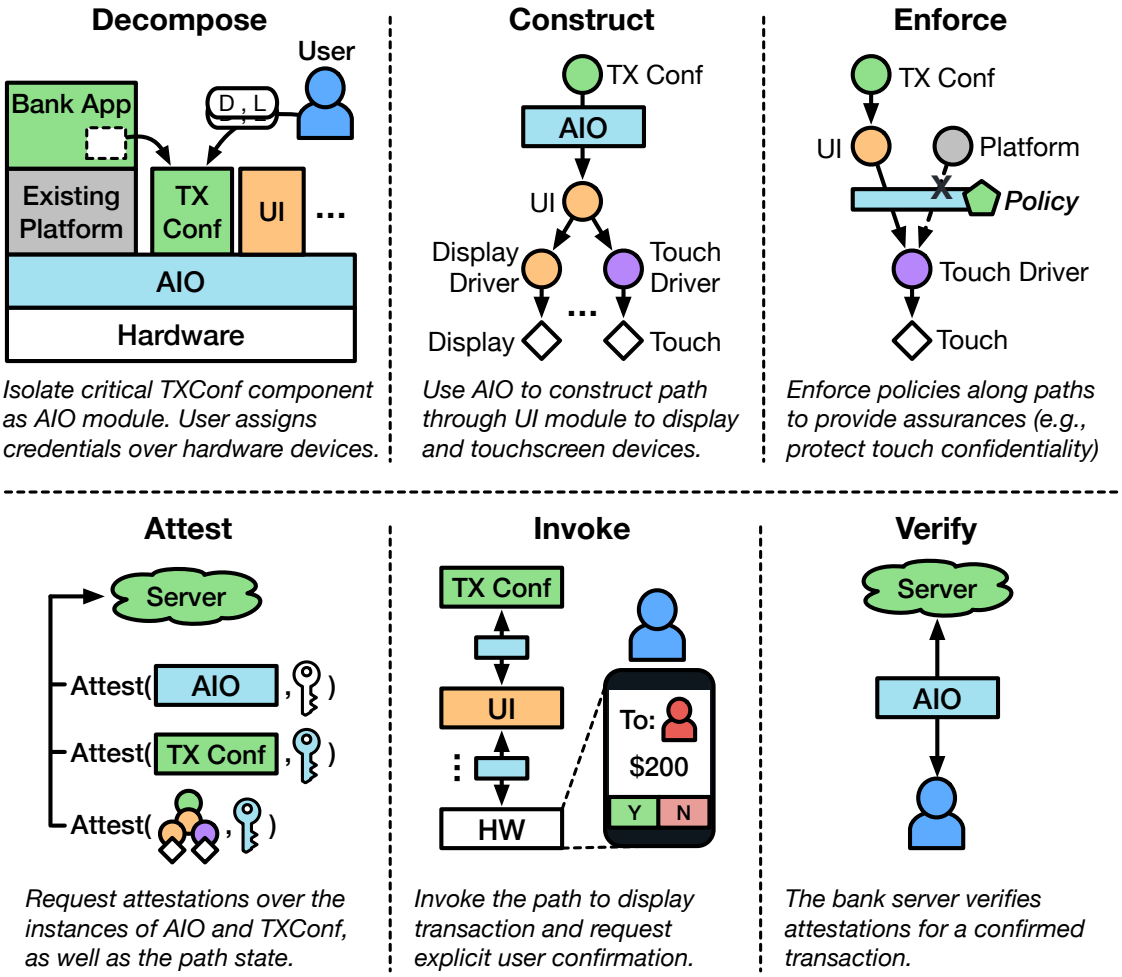


Figure 5.2: High-level overview for accountable paths and AIO, discussed in the context of a banking application that wants to support explicit confirmation transaction from the user to protect against compromised applications (e.g., keyboard) and/or the platform software (including the OS). The steps at the top denote a common workflow.

Decompose In this example, the bank wants to use AIO to get stronger assurances for monetary transactions. In particular, the bank wants assurance that the user themselves explicitly confirmed the transaction, instead of being generated as a result of some malicious keyboard application or a compromised OS. The bank isolates this critical transaction confirmation (TXConf) component within a AIO module. When installing the bank app, the user delegates credentials to the TXConf module over the display and touchscreen devices at a high privilege level (due to the sensitivity of the data involved). The bank app can communicate with the TXConf module through AIO, which it will use whenever there is a transaction that requires explicit confirmation from the user.

Construct When the TXConf module runs, it interacts with AIO to construct an accountable path through an interface provided by the UI module (provided by an AIO toolkit) to the underlying display and touchscreen devices. The modules along the path are bound at runtime by AIO based on the most-privileged implementations available for the respective interfaces; these modules may be provided by multiple principals (as shown by the different colors). Depending on the bank's trust assumptions, the bank might choose to supply all the modules along this path (such that TXConf talks directly to the hardware), or rely on trusted modules, such as the UI module in this example.

Enforce At this stage, the accountable path constructed between TXConf and the underlying hardware devices consists of all the functionality, but it doesn't

provide the necessary assurances. In AIO, policies are used to obtain these assurances. A policy can be applied to a specific interface, which will subsequently be enforced when less-privileged modules interact through the interface. Policies in AIO are expressed as functions that take the interaction as input, and the decision (allow/deny) as output; policies may optionally modify the interaction itself (e.g., changing data) or even invoke other accountable paths. Here, the bank wants to protect the confidentiality of the user's touch input from less-privileged principals (such as the untrusted platform software). The bank applies a policy at the touch driver interface to deny access to the published touch events for any module with a lower privilege level than that of TXConf's credential over the touchscreen. TXConf will also use policies to ascertain the provenance of a touch event, which must come from the touch device itself (i.e., hardware interrupt from user physically touching the screen).

Attest The next step is to interactively attest to the bank's remote server that both the functionality (modules) and assurances (policy) are installed correctly. AIO provides this by attesting to: 1) its own state (a specific version of AIO is loaded and running), 2) the state of the TXConf module (a specified instance TXConf is executing within AIO), and 3) the state of the path that TXConf has established. The bank's server can verify these remote attestations based on a chain of trust that AIO roots in the device hardware. These attestations are generated as part of an interactive protocol over a secure channel established between TXConf and the server, such that the server can supply nonces for freshness.

Invoke After requesting the attestations, TXConf can now use the path by invoking the various interfaces exported by the UI module to display the details of the transaction and wait for the user to confirm (or deny) the transaction via the Y/N buttons. AIO mediates the interactions between the isolated modules along the path, enforcing policies over these interactions when required. AIO also provides mechanisms to reason about the attested state of the path across an entire sequence of interactions, over which the assurances must hold.

Verify If the user confirms the transaction, the TXConf module sends the transaction information along with the attestations from the earlier stage to the server for verification. The server processes the transaction only if the attestations verify.

5.3 Design of AIO

In this section, we describe the design of the AIO enforcement layer. At a high-level, AIO acts as an extensible reference monitor that mediates all interactions in between modules, as well as between modules and hardware device resources.

5.3.1 Chain of Trust

We present the chain of trust for AIO in Figure 5.3, which explains the roots of trust in the system and how trust is derived to enable secure booting, our credential scheme, and (remote) attestations. AIO relies on three asymmetric key pairs that form the roots of trust in the system: the *device key* (P_D, K_D), the *user key* (P_U, K_U), and the *attestation key* (P_A, K_A). The device manufacturer provides the device

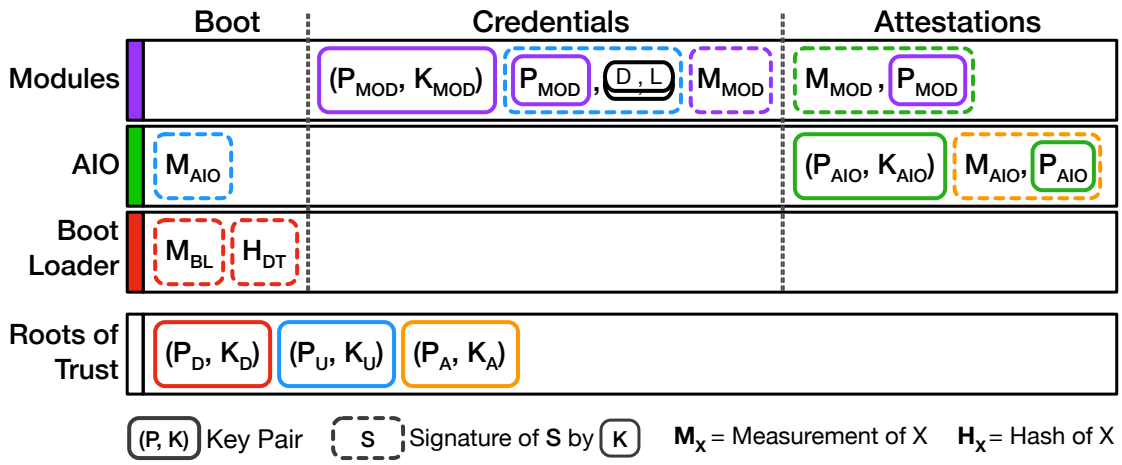


Figure 5.3: The chain of trust for AIO, broken down into the elements related to secure boot, credentials, and attestations. The device (D), user (U) and attestation (A) key pairs form the roots of trust.

key and the attestation key, as well as an initial version of the user key (which the user will update later). P_D, P_U, and (P_A, K_A) are stored in secure, tamper-proof, non-volatile storage which is commonly available on modern hardware; K_D and K_U are not stored on the device.

Secure boot relies on both the device key and user key. We assume there is a boot ROM that is responsible for loading and executing a second-stage bootloader; both of these are provisioned by the device manufacturer and trusted. The boot ROM uses P_D to verify: 1) the measurement of the bootloader prior to executing it, and 2) the hash of the trusted specification of the hardware (DT). The bootloader uses P_U to verify the measurement of AIO before executing it. At this point, we have a version of AIO that the user trusts loaded onto the system prior to the (untrusted) platform software.

The user key is used as part of our credential scheme to delegate access rights to other principals in AIO, which we discuss next. The attestation key is used for

bootstrapping our remote attestation scheme (discussed later).

5.3.2 Credentials

The credential scheme is backed by a trusted specification of the hardware that describes the full set of hardware resources present on the device and how they are interconnected. We rely on the device tree (DT) [103], as described in Section 3.3 to serve as the basis for our hardware specification, and introduce several new properties that are necessary for AIO. The *aio-deps* property contains the names of all non-standard properties that capture any dependencies on other devices; the set of standard properties includes *interrupts* and *clocks*. The *aio-prot* property associates the device with a particular hardware protection domain; on the i.MX6 SoC, this would reference the CSU with a resource specifier that contains the CSL register index. The *aio-class* property for each peripheral device to associate it with one or more high-level classes of devices (e.g., "Touchscreen"). The *aio-class* property, along with the *compatible* property, can be used in capabilities to specify devices.

5.3.2.1 Credential Scheme

AIO mints unforgeable capabilities at runtime and assigns them to modules; the modules use these credentials in building accountable paths. A capability (D, L) provides the right to interact with device D at priority level L. The device D refers to a device node described in the trusted hardware specification. The priority

level L encodes privilege levels, where L is an integer value in $[-\infty, \infty]$ with lower values corresponding to higher privileges.

A capability (D, L) can be used to derive a less-privileged capability (D', L'), where D' must either be D itself or a child of D as specified in the device tree hierarchy, and $L' \geq L$. The most-privileged capability is (/, $-\infty$), as it consists of the root of the hierarchy (/) and priority value ($-\infty$).

A capability (D, L) may be assigned to a principal P via a credential $[P_P, (D, L)]_{K_{P'}}$, where P_P is the public key for P. The credential is signed by $K_{P'}$, which is the private key for another principal P'. In order for a credential to be valid, the capability (D, L) must be derived from one of the capabilities assigned to the signing principal P'. We bootstrap the credential system with self-signed version of the most-privileged credential for the user: $[P_U, (/, -\infty)]_{K_U}$.

A capability (D, L) also implies the right to interact with all devices that D depends on, as per the hardware specification. These dependencies are typically expressed over a subset of the resources (e.g., UART controller depends on IRQ 30). We also allow capabilities to be directly delegated to a specific set of resources for a device by annotating D, which is useful for cases such as the user assigning an application a particular color for the notification LED (see Section 5.5.1).

5.3.3 Modules

A module consists of executable code and static, pre-initialized data. At run-time, modules may allocate memory within a fixed sized heap. A principal may

associate one or more credentials $C_i = [P_{Cred_i}, (D, L)]_{K_{Sig_i}}$ with a module by computing signatures over the module's code and data using K_{Cred_i}, which AIO can verify with P_{Cred_i}. After verifying the signatures, AIO mints and assigns the capabilities to the module. All interactions between modules, and ultimately with hardware devices, are mediated by AIO.

We treat all untrusted software, such as the untrusted OS and any applications it hosts, as if they are logically contained in a single module Module_{UT} that is assigned (/, ∞). While this capability allows the module to derive capabilities for any particular device (since / is the common root), it may never derive a capability with a higher privilege level than ∞ (which is the least-privileged level). This module may export default implementations of interfaces through AIO to other modules (which can be overridden by a more-privileged implementation), and likewise bind to interfaces implemented by other modules (subject to all policies on that interface).

5.3.4 Programming AIO

The I/O stack in AIO is composed of communicating isolated modules, similar in spirit to Binder IPC [110] and prior research on implementing a path abstraction [101, 102, 111]. At a high level, modules interact using function calls akin to RPCs; underneath, AIO implements a message passing architecture and enforces policies over inter-module communication.

Table 5.1 lists the API calls AIO provides to modules along with a high-level description of what they do. `Call`, `Publish`, `Wait`, and `Reply` represent a

standard API for communication via message passing. `Assign` and `Derive` are used in managing the set of capabilities for a module, either minting new capabilities through assigned credentials or deriving new, less-privileged capabilities. Next, we focus on the `Export`, `Bind`, `Apply`, `Enforce`, and `Attest_*` calls.

Function	**Description**
`Export`(Intf, Cap) → Desc/Error	Exports an implementation of an interface
`Bind`(Intf, Cap) → Desc/Error	Binds to an existing interface implementation
`Apply`(Intf, Cap, Filter) → Desc/Error	Applies a policy to an interface
`Call`(Desc, MsgIn) → MsgOut/Error	Invokes the bound function implementation and returns the output
`Publish`(Desc, Msg) → Error	Publishes a message to an event subscriber
`Wait`(Descs) → (Desc, Msg, Cap)/Error	Waits for an incoming event on a set of descriptors
`Reply`(Msg) → Error	Replies to the caller of a function with the computed output
`Enforce`(Act [, Msg]) → Error	Enforces a policy by specifying the action and optional data modification
`Assign`(Cred, Signature) → Cap/Error	Assigns a credential at runtime via signature over the module's code
`Derive`(Cap, D, L) → Cap/Error	Derives a new capability of (D, L) from an existing capability
`Attest_AIO`(Nonce) → Quote	Returns an attestation over the initial state of AIO
`Attest_Self`(Nonce) → Quote	Returns an attestation over the initial state of the calling module
`Attest_Path`(Desc, Nonce [, Pin]) → Quotes	Returns attestations over the modules along the specified path

Table 5.1: API functions that AIO provides to modules. Abbreviations: Cap = Capability, Cred = Credential, Desc = Descriptor, Filter = Policy Filter, Intf = Interface, and Msg = Message.

5.3.4.1 Export and Bind

`Export` allows modules to provide implementations of named/typed interfaces that other modules will construct paths to via `Bind`. The capability provided to `Export` serves to restrict the set of modules that can `Bind` to the implementation. Given a `Bind` request with capability (D, L), AIO attempts to construct a path to the most-privileged implementation that matches (D, $*$). Note that these bindings may change over time as the set of implementations changes; if a new, more-privileged implementation is introduced then the binding module will receive a revocation event and subsequently call `Bind` again.

AIO supports synchronous and asynchronous interfaces. For a synchronous interface, the calling module will invoke the implementation via `Call` with the input parameters; the exporting module receives the call via `Wait` and responds via `Reply`. For an asynchronous interface, the exporting module receives subscriber descriptors via `Wait` and may later `Publish` events to subscribers (who receive the events via `Wait`).

When a module receives an call event for an exported interface via `Wait`, it can modify its behavior based on the capability of the caller (likewise for subscribers). The implementation may use the privilege level of the caller to, for instance, multiplex requests from multiple callers where the most-privileged caller gets the highest priority. Additionally, if the capability of the caller is tagged with a resource, the callee may interpret this resource information to constrain the events it publishes to the subscriber for an asynchronous interface, or to deny (or modify) accesses for

a synchronous interface. For a more concrete example, consider an application providing an interface to the set the color of the notification LED. Credentials for the LED may be associated with resource information describing the set of valid colors; if not tagged with a resource, all colors are valid.

5.3.4.2 Apply and Enforce

Modules can apply a policy to a named/typed interface by calling `Apply`. The capability provided to `Apply` serves to restrict the set of paths this policy applies to. Given a policy with capability (D_P, L_P) and a binding module with capability (D_B, L_B), the policy will be applied if $L_P \leq L_B$. Policies may also be applied if they are assigned a resource-tagged capability. Given a binding with capability ($D_B.R$, L_B) for the same resource, the policy will be applied if $L_P \leq L_B$. Given a binding with capability (D_B, L_B), the implementing module must determine whether or not the policy should apply on a per-invocation of the interface by interpreting the resource information R. The filter argument provided to `Apply` specifies whether the policy should apply on the `Call` and/or `Reply` sides of the interface.

In handling a `Call` (or similarly a `Publish`) that a given policy applies to, AIO issues policy enforcement requests that the policy module receives via `Wait`. In response to such a request, the policy module invokes `Enforce` to let AIO know whether the interaction should be allowed or denied, and may optionally modify the data involved in the interaction. By default, AIO issues an enforcement requests in order of privilege from most to least; this enables more-privileged policies to deny

the interaction before less-privileged policies can operate over it. However, AIO also provides a facility which allows more-privileged policies to execute last as well. This mechanism is similar to pre- and post-hooks in event-based systems.

Assurances Policies are extremely powerful components in AIO, that not only enable access control but are also used to obtain assurances such as data confidentiality and integrity.

A module may protect confidentiality along a path by applying a policy with a certain privilege level L that denies access to the interfaces by any less-privileged module. The application of this policy may fail in the case that a module providing functionality along the path is less-privileged; when AIO returns this error, the module can choose whether it is willing to lower the level of confidentiality protection (or not).

A module may also protect integrity of data along a path against unauthorized modification. In practice, there are many ways to achieve integrity assurance, and AIO is flexible in supporting a variety of different mechanisms. First, a module may consider integrity to be maintained as long as all modules along a path are trusted. These trust assumptions of the module may be specified in the form of hashes of modules whose implementations are trusted to be correct, or (more broadly) in the form of public keys associated with principals that are trusted to correctly implement and provision modules. In other cases, it may not be possible to enumerate (or fully satisfy) the set of trust assumptions, potentially as a result of trying to maximize re-use of existing, untrusted code. Therefore, the module can employ policies that

apply to the inputs and outputs of any untrusted modules along a path to enforce behavior specifications over their accesses.

5.3.4.3 Attestations

AIO supports establishing a secure channel between a module and a remote server, which can be used for communication and as part of the remote attestation protocol. While the remove server should be able to authenticate a running AIO instance, the server should not be able to learn a unique identifier that can be used to link AIO instances over time. To this end, we rely on a group signature scheme similar to Intel EPID used in SGX [43], in particular the Boneh-Boyen-Shacham short group signature scheme [112]. The attestation key pair (P_A, K_A) consists of the group public key P_A for all of the manufacturers' devices, while K_A corresponds to a per-device private key. The remote server will learn that a device belongs to the group of the manufacturers' devices (through a certificate chain that certifies P_A), but not the particular device.

Communication between modules and remote servers are relayed by a non-secure network proxy; we use standard key exchange protocols to establish shared keys between remote services and secure modules.

Attesting to AIO The most basic attestation is over the initial state of AIO itself. While loading and verifying the AIO image, the trusted bootloader computes a measurement M_{AIO} over the initial state of AIO. The bootloader generates a new ECDSA key pair (P_{AIO}, K_{AIO}) and certifies the public key by signing a message

$[M_{AIO}, P_{AIO}]_{K_A}$. The bootloader passes the key pair and the certificate to AIO for use in higher-level attestations. This scheme follows from the key derivation and attestation approach in RIoT [113], but modified to make use of group signatures.

A module may request an attestation of AIO via the `Attest_AIO` API call, passing in a nonce. In response, AIO returns the signed message from the bootloader, which certifies the public key P_{AIO}, and a signature over the nonce using K_{AIO}. The verifier has knowledge of the P_A associated with the device, which it uses to verify the certificate for P_{AIO}, which proves the existence of a trusted boot chain to load AIO. The verifier will then check whether M_{AIO} included in the certificate corresponds to a known, trusted instance of AIO; if so, the verifier uses the included P_{AIO} to finally verify the signature of the nonce.

Attesting to a Module A module may also request an attestation over its own initial state to send to the remove server. When AIO loads modules, it computes and stores a measurement M_{MOD} of the initial state of the executable code and static data; the credentials assigned to the module are not included in this measurement. A module may request this attestation via the `Attest_Self` API call, which returns $[M_{MOD}, \text{Nonce}]_{K_{AIO}}$. The verifier uses the P_{AIO} certified by the attestation to AIO to check the signature of this message, and also checks whether M_{MOD} corresponds to a trusted instance of the module.

Attesting to an Accountable Path AIO provides attestations over the full path between a module and underlying hardware resources. A module may invoke

the `Attest_Path` API call to request attestations over all modules along a path specified by a descriptor, which returns a signature over a graph data structure G which represents the path $[G, \text{Nonce}]_{K_{AIO}}$. The graph structure consists of nodes corresponding to the modules (and in particular their measurement); the edges represent a binding from one module to another for a particular interface. The verifier uses the P_{AIO} certified by the attestation to AIO to check the signature of this message, and also checks the complete state of the path.

Note that these path attestations capture the instantaneous state of the system at the time of the request. However, it is typically necessary to reason about the state of the path *across* some sequence of operations; for example, the expected path for a trusted UI is maintained while the user interacts with the interface. To this end, a module may optionally supply the "Pin" parameter to instruct AIO that the module wants to be notified if the path deviates from the attested state. Later, the module can unpin the path by invoking `Attest_Path` again but with an "Unpin" parameter.

5.4 Implementation

We rely on the ARM TrustZone hardware security extensions for our prototype implementation. Figure 5.4 shows an overview of the important components of the hardware platform, along with how we organized the software to augment the existing Android/Linux OS with support for accountable paths. AIO runs in the secure world, acting as both the secure kernel and the secure monitor, while the

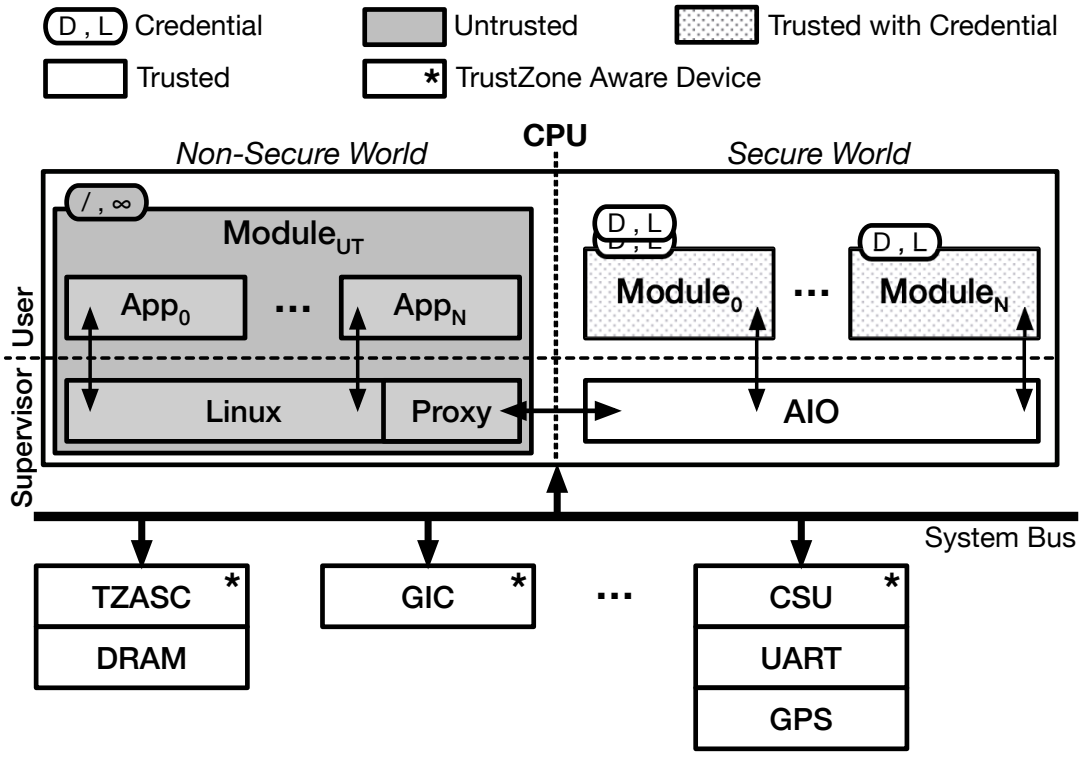


Figure 5.4: Overview of the important components of the hardware platform and organization of software involved in augmenting Android/Linux with accountable paths via AIO .

existing Android/Linux OS and applications run in the non-secure world. AIO executes all modules in isolated, trusted execution environments in secure world user mode. To expose accountable paths to the non-secure world, we created a Linux kernel module that acts as an untrusted proxy to AIO; the proxy communicates with AIO via the (`SMC`) interface, where AIO provides a similar API to that described in Table 5.1.

Our implementation of AIO is very loosely based on OP-TEE [60], an OS for the ARM TrustZone secure world that provides execution environments for trusted applications that are invoked by the untrusted OS. We primarily make use of OP-TEE support for configuring the MMU, loading ELF binaries, secure monitor handlers, and drivers for the GIC and serial port devices.

5.4.1 Hardware Protections

AIO relies on hardware protection mechanisms to mediate device accesses and to isolate itself from the rest of the system. The TrustZone Address Space Controller (TZASC) mediates accesses to RAM, and can be configured to allow or deny accesses to regions according to the world and type (R/W). AIO configures the TZASC to protect the regions of memory occupied by AIO (and all loaded modules) from all non-secure world accesses. The Generic Interrupt Controller (GIC) is a TrustZone aware device that enables the secure world OS to assign specific interrupts as either secure world or non-secure world, and limits the priority of non-secure world interrupts (e.g., to avoid DoS attacks). Finally, the Central Security Unit (CSU)

mediates accesses to MMIO devices. This is a manufacturer-specific IP block from NXP, which is found on the hardware platform we use in our evaluation; however, similar functionality is provided in most TrustZone instantiations. The secure world OS can configure the CSU to allow or deny accesses to devices based on the world and mode; optionally, when the access is denied, the CSU can be configured to deliver the fault to the secure world monitor. These latter two protections are used by our built-in modules, which we discuss in Section 5.4.2.

5.4.2 Modules

AIO executes all modules in isolated, trusted execution environments in secure world user mode. AIO loads modules that come in the form of signed ELF binary images along with a set of credentials. AIO verifies that the credentials are valid and then uses the public keys bound to the credentials to verify the signatures over the ELF image. Each module runs in a separate address space in secure world user mode, and communicates with AIO via the API exposed through the AIO system call interface (see Table 5.1).

Inter-Module Communication AIO enables inter-module communication with an interface description language that is similar to seL4 [98]. Each interface specifies a message that consists of a set of values (i.e., data words) and a set of items. Items support transferring a variable-sized data buffer or sharing a memory mapping for a region that contains the data. A buffer item consists of a region of memory in the sender's address space which is copied to a region in the receiver's address space. A

mapping item consists of a region of memory at the page granularity which will be granted to (or shared with) the receiver. The specification for a buffer item consists of the maximum size of the buffer to be sent or received, and (if receiving) the address of the buffer in the module's address space. The specification for a mapping item consists of the maximum size of the mapping to be sent or received, and (if receiving) the required access rights for the mapping (i.e., read or write).

AIO maintains two queues of messages for each module: one consisting of empty messages that can be acquired, and one consisting of filled messages to be returned from future `Wait` calls. AIO maps each message into both the module's address space as well as AIO's own address space in order to support efficient transfers from one module to another, without the need to use a temporary in-AIO buffer. Each thread for a module is assigned a single message at a given time. A thread begins execution at the "AIO_entry" point, which passes in the message assigned to the thread. When a thread calls `Wait` and there is an available filled message, the call immediately returns with the message; Otherwise, the thread attaches itself to its module's wait queue until a message is filled. When a thread invokes a `Call` on a bound descriptor, AIO identifies the next module in the call path and attempts to acquire a message; if successful, AIO transfers the message to the destination module, places it on the filled queue, and notifies a waiting thread to run (if any). If unsuccessful, the thread attaches itself to the destination module's wait queue until an empty message is supplied to be used in the transfer.

Built-in Modules For modules to interact with hardware devices, AIO must provide some platform-specific, built-in modules upon which higher-level drivers can be written; for our platform, the set consists of a memory-mapped I/O (MMIO) module and a interrupt controller (IRQ) module. The MMIO module exports interfaces to read and write device registers (e.g., "MMIO_Read32"). The IRQ module acts as a driver for the primary interrupt controller on the CPU, handling IRQ and FIQ exception modes and exporting interfaces for managing IRQs to other modules (e.g., "IRQ_Enable"). These modules hold the most-privileged capability for their device.

To reduce overhead, the MMIO module provides a "MMIO_Map" interface that allows a calling module to request that a particular physical address region be mapped into their address space. While "mmaped" accesses are extremely efficient, they do not inherently support interposition of policy as they side-step the path interfaces. Therefore, when a policy is applied to the read or write interface, AIO must configure the hardware CSU protections to force such accesses to generate faults that AIO can subsequently handle (similar to SeCloak [1]). In handling the fault, AIO emulates the instruction that caused the access fault, invoking the path for the MMIO access function and enforcing the applied policy.

5.4.3 Accountable Paths with Linux

In our implementation, we retrofit Linux/Android with support for accountable paths by exposing AIO through a proxy Linux kernel module. One of the primary goals is to enable AIO modules to make use of the existing platform soft-

ware to minimize their TCB. This is useful when modules do not require assurances from certain parts of the stack, or are able to provide these assurances at the ends (e.g., TLS channel over untrusted networking stack).

Additionally, we expose accountable paths to Linux via "shims" which allow the existing platform software to make use of functionality provided by more-privileged modules in AIO. We describe the operations of shims through an example for the touchscreen, in which a driver is loaded as a secure module in AIO. The driver module applies deny policies to its lower-level interfaces (i.e., I2C and IRQ) to protect against their use by other, less-privileged software. These policies prevent the (untrusted) touchscreen driver in Linux from making accesses; however, assuming there is no policy in place to prevent access to the touch events, Linux should still be able to consume them. Therefore, the Linux-resident touchscreen shim subscribes to the `Touch_Event` interface in AIO, and also registers itself as an touchscreen device to the input subsystem in Linux (in place of the existing touchscreen driver). Whenever the shim receives a published event, it marshals the data and appropriately signals the input subsystem that a touch event occurred; the applications built on top of the input subsystem are unaware, as they still access the touch data via the standard sysfs interface (e.g., `/dev/input/event0`).

5.4.4 Policy Interposition

When modules interact through an interface with applied policies, AIO issues enforcement requests to the modules that applied the policy. Given the fact that

multiple policies from different principals might apply to a path invocation, the overhead of message copying and context switching may be significant. Therefore, we allow policies to be expressed as extended Berkeley Packet Filter(eBPF) [95] programs that can be verified, just-in-time compiled[2], and executed within AIO itself. The program context for AIO is similar to that of traditional XDP BPF programs, which provide a very low-level view of packet data by specifying pointers to the start and end of the packet data buffer. In our case, AIO passes in a context parameter that specifies the start and end of a "serialized" message which contains the core message structure concatenated with any data contain in buffer items.

The header of the eBPF image specifies the singular interface that the policy applies to and includes a set of credentials for the policy. Each invocation of the eBPF program constitutes a separate enforcement request by AIO over communication through that interface. Unlike policies implemented by full-fledged modules, eBPF-based policies apply to a single interface and cannot construct/invoke paths to other modules while handling an enforcement request. We support a simple tuple space data structure for storing data across executions.

5.5 Use Cases

We begin by discussing a toolkit that we developed for building secure applications, followed by different application scenarios and how they might be realized in AIO.

[2]Currently, verification and compilation are performed outside of AIO.

5.5.1 Toolkit

We provide a set of (trusted) modules that serve as building blocks for applications. Unlike commercial TEE solutions [16–19], our set of building blocks is not limited to a fixed set that is only provided by a single principal, and may be replaced/extended to fit each application's trust assumptions. Next, we discuss several of these modules and the properties they provide.

5.5.1.1 Remote Proxy

Often a principal might only require a minimal secure module running on the user's device that interacts with software running on a (trusted) remote server owned by the principal. The "Remote Proxy" toolkit module allows a remote server to interact with AIO as if the remote software was running as local module.

Initially, the proxy module is loaded by supplying one or more credentials that may be used for networking, along with an input specifying a network endpoint for the remote server. The proxy and remote server participate in an interactive protocol, such that: 1) the proxy provides attestations over the measurements AIO and the proxy module itself, and 2) the server sends credentials with the necessary capabilities and signatures over the measurement of the proxy. The proxy module passes these credentials to AIO via the `Assign` call, which will mint and assign the appropriate capabilities to this instantiation of the proxy module. Afterwards, the remote server sends commands to the proxy module which mirror the API calls between local modules and AIO (see Table 5.1). The proxy module is responsible

for invoking the corresponding system calls on behalf of the remote server, allowing the server to construct and invoke paths, apply policies, and request attestations over system state.

5.5.1.2 Display Composer

The "Display Composer" toolkit module serves to securely compose multiple, individual display layers from various modules to generate a final framebuffer that is rendered to the user via the display. The composer looks at the calling module's capability to determine its privilege level. The composer uses these privilege levels to serve as the Z-axis values for ordering the layers: more-privileged layers are displayed above less-privileged layers, with the untrusted layer acting as the background.

5.5.1.3 Notification LED

The "Notification LED" toolkit module provides an interface for setting the color of the notification LED to unambiguously notify the user that they are interacting with a trusted application. This prevents an attack whereby a malicious application could spoof the UI of a trusted application, potentially allowing an attacker to learn sensitive information the user would enter (e.g., banking authentication information). Given a credential for the LED device, a module may use this building block to turn on/off the LED; the module may also prevent other, less-privileged applications from configuring the LED by applying a policy that denies such accesses. When delegating credentials to various applications, the user

may assign each application a specific color (or part of the spectrum) by tagging the device in the credential with a resource specifier that encodes the whitelist of colors. The LED module looks at the calling module's capability to check for this tag; if present, the LED module checks to see if the requested color value is in the whitelist.

5.5.2 Application Scenarios

Next, we discuss several different application scenarios, focusing on the desired assurances and how AIO can help to provide them.

5.5.2.1 Mobile Banking

Mobile banking applications provide rich functionality including the ability to deposit checks, transfer money, and view financial records; however, such functionality also makes these applications a prime target for attackers. Many components of these banking applications can benefit from being built on top of accountable paths, such as PIN input-based authentication and explicit user confirmation for transactions (as discussed in Section 5.2).

When the user opens the banking application, they must authenticate themselves to the bank's remote server. This typically involves the application requesting a PIN (or password) from the user and sending it to the remote server for validation and establishing an authenticated session. It is important to protect the confidentiality of the user's PIN against attacks from malicious software or a compromised

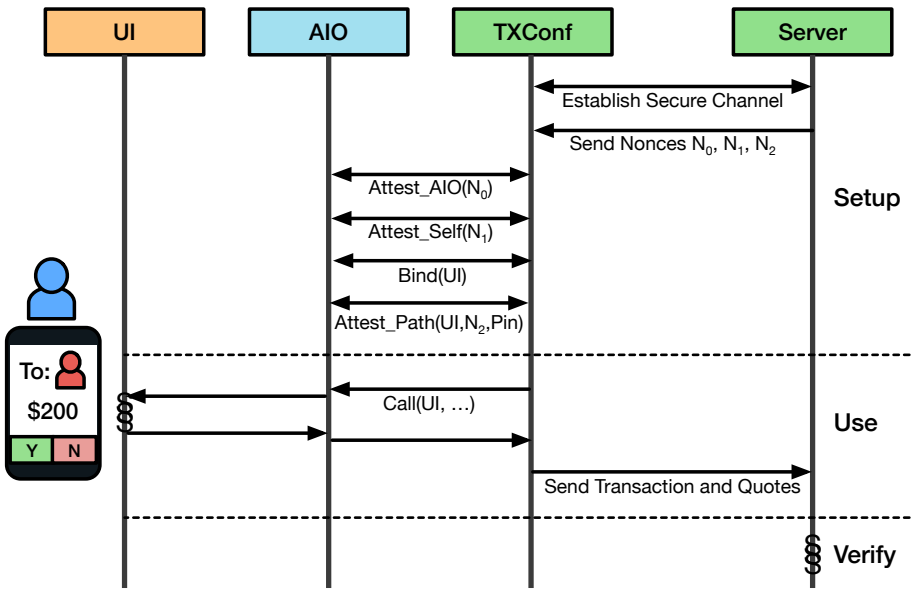


Figure 5.5: Overview of how the Bank TXConf module uses AIO to support explicit confirmation of user transactions.

OS; however, it is not necessary to protect integrity, since an incorrect PIN will simply cause the authentication to fail. The authentication module must apply policies that deny any less-privileged modules from snooping on the entered PIN (or lower-level touch events).

Figure 5.5 presents an overview of how explicit user confirmation for transactions may be realized on top of AIO. The TXConf module establishes a secure communication channel with the remote server, via TLS over a TCP connection via an untrusted network stack (not shown). The TXConf module begins by requesting attestation quotes over the initial state of AIO and the module itself; the nonces from the server are used to ensure freshness of these quotes. Afterwards, the TXConf module constructs and attests to a path to the interface exposed by the UI module, which is included in the toolkit. In order to make sure that this attestation holds across the invocation of the path, the TXConf module requests that this

path be pinned. Next, the TXConf module invokes the path to configure the user interface for displaying the transaction details and to wait for the user to confirm (or deny) the transaction. If the user confirms the transaction, the TXConf module will send the transaction information and attestation quotes to the remote server. If the quotes are valid, the server will execute the transaction.

5.5.2.2 System Integrity Policy

CLKSCREW [114] is one example of an attack on system integrity, involving the misconfiguration of the CPU voltage and frequency regulators in order to cause faults in secure world computation (which can even lead to the exposure of cryptographic secrets). With AIO, we have the opportunity to provide a reliable, software-based defense via installation of a straight-forward policy that mediates accesses to the CPU voltage and frequency scaling registers by less-credentialed software (e.g., Linux CPUFreq driver). We present pseudocode in Algorithm 1 which captures how such a policy module can be implemented in AIO, using the API described in Table 5.1. We assume the policy module has credentials over the CPU voltage and frequency regulator device(s) that are accessed via MMIO.

The module applies a policy over calls to "MMIO_Write"; since this is an access control policy, we only need to register for the first execution phase for the policy filter. In addition, the module binds to "MMIO_Read" in order to read the current values of the frequency and voltage regulators. When AIO issues a policy enforcement request for "MMIO_Write" to the module, the module checks if

Algorithm 1 Pseudocode an AIO policy module to protect against CLKSCREW [114] attacks that target system integrity.

```
1: function GetAction(volt, freq)
2:     return Allow if valid pair, otherwise Deny
3: function Module_Entry()
4:     rIntf = { "MMIO_Read", inVals = 1, outVals = 1 }
5:     rDesc = Bind(rIntf, 0)
6:     wIntf = { "MMIO_Write", inVals = 2 }
7:     wDesc = Apply(wIntf, 0, FILTER_CALL_INPUT)
8:     loop
9:         desc, msg = Wait()
10:        if desc == wDesc then
11:            addr, value = msg.values
12:            action = Allow
13:            if addr == VOLT_ADDR then
14:                freqMsg = Call(rDesc, { FREQ_ADDR })
15:                action = GetAction(value, freqMsg.values[0])
16:            else if addr == FREQ_ADDR then
17:                voltMsg = Call(rDesc, { VOLT_ADDR })
18:                action = GetAction(voltMsg.values[0], value)
19:            Enforce(action)
```

a relevant device register is being addressed; if yes, the module checks if the resulting (voltage, frequency) point would exist within the safe operating limits for the device.

We implemented and applied this policy on our hardware platform, deriving the functions that describe the highest voltage allowed for each frequency (and vice-versa) based on the hardware parameters. In order to test this policy, we instructed the Linux cpufreq driver to try to set the voltage and frequency regulators beyond the limits, which the policy denies. Additionally, we instructed the Linux cpufreq driver to set values within the valid operating limit (and read the current values), which the policy allows.

5.5.2.3 Geofencing

Let's consider a geofencing application that collects GPS sensor samples to determine the set of policies to apply and enforce at any given time, where each

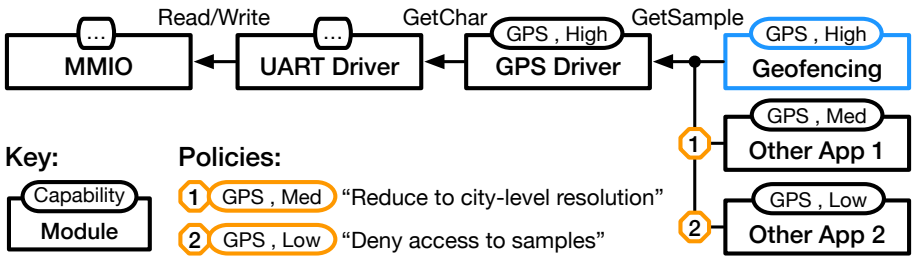


Figure 5.6: An example set of accountable paths for a geofencing app (and other apps) that use GPS location sensor samples. The user delegates capabilities with different privilege levels to the applications. The user can apply policies on the paths, such as reducing the resolution of (or denying access to) the GPS samples.

policy is associated with a specific geographic region. We show the modules and accountable paths involved in Figure 5.6. The GPS driver exposes a "GetSample" interface to other modules that returns standard information such as (lat,lon) coordinates. The GPS driver communicates with the hardware GPS radio device via a serial (UART) bus, using standard NMEA protocol; the UART driver communicates with the hardware bus controller via memory-mapped I/O (MMIO). Let's assume these drivers are provisioned by the device manufacturer.

When the user installs the geofencing application, the user assigns the application principal a capability for the GPS device with a high privilege level. The geofencing module will use this capability to construct an accountable path to the "GetSample" interface associated with the GPS device. When constructing the accountable path, the geofencing module specifies its security properties and trust assumptions. In this particular use case, the module does not require confidentiality along the path; however, the geofencing application provider may only trust the device manufacturer's implementation of the GPS driver to be correct. In this case, the device manufacturer's driver is the most-privileged implementation running, so

the platform successfully constructs the path.

The user may wish to install policies to reduce the resolution of the GPS sensor samples for other, less-privileged applications. The user can delegate a more-privileged capability for the GPS sensor device to a policy which will apply to the "GetSample" interface and modify the resulting data, such as reducing the resolution of the coordinates to city-level granularity by dropping some number of decimal places. As shown in Figure 5.6, multiple policies may be applied at varying privilege levels. In this case, there are two policies that affect the data along the path for the less-privileged applications.

5.6 Evaluation

In our evaluation, we aim to demonstrate the practicality of our approach and implementation by measuring the performance of AIO operations. All evaluations are performed on the Boundary Devices Nitrogen6Q single-board computer, which contains an i.MX6 SoC with a quad-core 1 GHz ARM Cortex A9 processor and 1 GB of DRAM. We used the BD070LIC2 touchscreen display, which connects to the system via an LVDS interface for the display and an I2C interface for the touchscreen. For our cycle measurements, we rely on the hardware cycle counting support from the performance monitoring unit; for our timing measurements, we use the EPIT timer which we configure to tick every 1 μS. For each experiment, we compute the statistics based on the results of 100 trials. While AIO has multi-core support, we run these experiments with only a single core enabled.

Type	C Src	C Hdr	ASM	Total
Overall				
AIO-Specific	4,274	1,077	426	5,777
Total	90,817	25,454	6,500	122,771
Drivers				
CSU	80	9	0	89
GIC	562	18	0	580
TZASC	146	143	0	289
Libraries				
libaio	60	169	38	267
libfdt	1,862	419	0	2281
libmbedtls	43,840	9,791	0	53,631
libutils	18,147	3,405	219	21,771

Table 5.2: Breakdown of the lines of code (LOC) for different parts of our AIO implementation. We list the LOC according to the language used (and source versus header) along with the total LOC.

5.6.1 Size of TCB

We first present results to quantify the size of the TCB for our implementation of AIO. In Table 5.2, we quantify the size of the TCB for our implementation of AIO by showing a breakdown for the lines of code. "Overall" consists of all code, including drivers and libraries; the AIO-specific portion includes all of the code except for platform-specific components, MMU handling, and ELF loading. The "Total" of 122,771 is a significant overestimate due to the LoC count including many files from OP-TEE and supporting libraries that are not compiled for AIO (e.g., trusted application (TA) support). "Drivers" consists of all driver code, which is further broken down into specific drivers that we added to (or borrowed from) OP-TEE. "Libraries" consists of all the libraries we use. We rely on libfdt for parsing the flattened device tree file, libmbedtls for cryptographic functions, and libutils for various utilities (e.g., standard library functions, bget allocator, tracing); we

introduced libaio that modules link with to invoke AIO's API functions.

5.6.2 Baseline

In Table 5.3, we show the results of experiments that measure the simplest forms of various operations to establish a performance baseline for AIO. We employ two benchmarking modules that each have a single thread and a message queue of size two; the SMC experiment uses a separate benchmarking module running in the non-secure world.

The SMC and SVC calls represent the overhead for communication between $\text{Module}_{\text{UT}}$ in the non-secure world and all secure world modules with AIO (respectively). SMC handling involves saving (and restoring) more register state. In the following results, we focus on the SVC interface (i.e., from a secure world module).

Next, we look at the cost to export, bind, and invoke paths to both a null function interface (with no inputs or outputs) and a null event interface (with no data). Beyond the basic invocations (i.e., call and publish), we also evaluate the case when an "Allow" policy is applied to the interface and enforced by AIO; we consider the cases when the policy is applied by a module and as a standalone eBPF program. Calling a function interface involves four context switches (essentially two SVCs), while publishing to an event interface involves only two. The higher standard deviation for the publish operations is attributed to the fact that the message queue is of size two for each module. Unlike call, publish does not need to block by default; however, if no messages are available in the subscriber's queue, then it will

		Cycles	
Operation	**Notes**	**Mean**	**StdDev**
Null SVC	Modules to AIO	657	23
Null SMC	NS World to AIO	828	105
Null Function			
Export		3,744	255
Bind		3,551	312
Call		6,746	739
Call+Policy	Allow	9,930	509
Call+Policy*	Allow (eBPF)	7,267	476
Null Event			
Export		3,520	205
Pub		6,101	3,146
Pub+Policy	Allow	8,912	5,988
Pub+Policy*	Allow (eBPF)	6,813	3,524
Derive		4,467	1,284

		Time (μS)	
Operation	**Notes**	**Mean**	**StdDev**
Attest_AIO		53,590	172
Attest_Self		53,587	160
Attest_Path		53,593	162

Table 5.3: Baseline performance results for various operations.

block until a message becomes available for use. Small policies expressed as eBPF programs improve performance substantially, by a factor of roughly 6x in the case of function interfaces (521 vs 3184 cycles); this is because AIO can execute eBPF-based policies safely without context switching to a different protection domain.

Attestations are expensive operations as they include an asymmetric cryptographic signature operation to generate a fresh quote given a nonce passed in by the calling module. In our implementation, we use the mbed-tls library [115] for an ECDSA signature scheme using the NIST P-256 curve. If many attestations are needed in practice, it is possible to reduce this overhead by exchanging a symmetric key for use in an HMAC-based attestation scheme.

Operation	Notes	Cycles Mean	Cycles StdDev
MMIO Write (Call)			
Baseline		3,859	517
Policy*	Allow (eBPF)	4,642	455
MMIO Write (Mapped)			
Baseline		35*	2
Policy*	Allow (eBPF)	4,839	342
Trap+Call		3,946	392
Trap		1,723	285

Table 5.4: Built-in MMIO module performance.

5.6.3 Built-in Modules and Drivers

As part of our implementation, we provide two built-in modules, MMIO and IRQ, that enable higher-level drivers to be written (see Section 5.4.2). We have also implemented a number of driver modules as part of our toolkit for a variety of different devices (e.g., GPIO controller, touchscreen). These drivers are functional ports from the corresponding Linux version, modified to run as modules on top of AIO. Here, we evaluate the performance characteristics for some of these built-in and driver modules.

Table 5.4 contains the results from experiments to evaluate the costs associated with applying policies at the level of MMIO operations. For these experiments, we used a benchmarking module running in the non-secure world that makes register accesses to the Clock Control Module (CCM) device. For the baseline mapped experiment, we measured the cycle count across ten writes to the register due to the cycle count inaccuracies we witnessed for single instructions. Calls to this interface require less cycles than calls to the NULL function interface in Table 5.3, since the MMIO module is built into AIO itself and therefore doesn't require additional

Operation	Notes	Time (μS) Mean	StdDev
Touch Event (Linux)		13,281	746
Touch Event (AIO)			
to Module		5,676	55
to Linux		7,046	6,692

Table 5.5: Touchscreen path performance.

context switches. As we can see, the common case where no policy is applied (see "Baseline") is significantly improved by issuing writes to mapped memory instead of invoking the "MMIO_Write" interface directly (35 vs 3,859 cycles). And, in the case policies are applied, issuing writes to mapped memory only incur a small overhead of less than 200 cycles (roughly 4%). We provide a breakdown of the costs associated with handling a write to mapped memory while a policy is applied to that region: "Trap" includes all work to handle the data abort fault and determine the access, while "Call" includes invoking the path to emulate the access.

Table 5.5 shows the results from experiments for the touchscreen driver to understand how the latency changes when running the components in AIO. Here, latency refers to the time between when the interrupt is raised on the GIC (as a result of a touch) and when the touch event is published. The "Linux" case corresponds to using the existing drivers in Linux; the "AIO" case substitutes the touchscreen, I2C, and GPIO drivers with ported implementations running as modules in AIO. We look at two sub-cases for whether the AIO module publishes touch events to: 1) a subscribing module, or 2) the Linux shim connected to the input subsystem. In the "Linux" case, we see that the latency for handling a touch event is roughly 13 ms. When implemented on AIO, we see that the latency drops to 5-7 ms. This difference

is largely due to the fact that RUNNING/RUNNABLE threads from AIO modules have higher priority than any threads from Linux; as long as AIO is active on a CPU core, it does not give control back to Linux until the idle thread is scheduled.

5.6.4 Summary

Our evaluation benchmarks various aspects of AIO and common I/O devices. Our results don't reveal any unexpected performance anomalies, and more importantly, show that AIO can be implemented on current hardware, and provide sufficient performance to support useful applications. Indeed, we have used the components we have evaluated (e.g., touchscreen path) to implement various end-to-end application applications, including the secure mobile banking scenario discussed in Section 5.5. All of these applications run stably with no perceptible change in usability.

5.7 Conclusion

In this chapter, I have introduced AIO, an isolated enforcement layer that enables transparency and control over the software stack for handling I/O data, benefiting both users and application principals. A key component of AIO is the new accountable path abstraction, which represents a software path between a set of hardware resources and a local or remote endpoint. This simple abstraction enables expressive control over how sensitive data is collected, processed, and shared, and also supports building end-to-end secure applications with proven assurances (e.g.,

confidentiality). Accountable paths handle multiple (mutually distrusting) principals, enabling fine delegation of trust and safe composition of software from different principals that may be contributing various parts of the I/O stack. Through our implementation efforts, we showed how to apply accountable paths to existing platforms such as Android/Linux, and we demonstrated the practicality of our solution.

Chapter 6: Conclusion and Future Work

In this dissertation, I motivated the need to introduce a new, isolated enforcement layer given the problems facing users with respect to a lack of control and assurance over how their sensitive data is being collected, used, and shared. My contributions include the design and implementation of two instantiations of such an enforcement layer: SeCloak and AIO. These contributions support the following thesis: *Introducing an enforcement layer between the hardware and system software can enable end-to-end secure applications while giving users fine-grained control over their devices.*

SeCloak addresses a key point in the control policy space for users: providing on/off control for peripheral devices. SeCloak runs as a platform-agnostic layer that provides the abstraction of secure, virtual switches that the user can reliably configure. I showed that it is possible to provide such reliable controls regardless of the correctness of the rest of the platform software, including the OS. Through the evaluation, I demonstrated low overhead of our trap-and-emulate approach for handling coarse-grained hardware protection domains.

AIO introduces a new "accountable path" abstraction that enables constructing and reasoning about the end-to-end I/O stack between application endpoints

and underlying hardware devices. Accountable paths allow for more expressive policies to be enforced over the software stack; these policies can be used to derive various assurances, which AIO provides proof of via attestations over (parts of) the accountable paths. I solved the problem of supporting many mutually distrusting principals that contribute various parts of the I/O stack. I demonstrated the utility of the accountable path abstraction through a variety of application usecases, and the practicality of AIO through performance benchmarks.

6.1 Future Work

While SeCloak and AIO solve core, technical problems, they also open up some additional research questions, as well as questions to address in the context of practical deployments of these systems.

6.1.1 Strengthening Foundations of Trust

A key goal of my work on AIO is to provide assurances over the handling of sensitive I/O data; however, these assurances naturally rest on any assurances over the correctness and security properties of AIO itself. Similarly, in SeCloak, the assurance that devices are disabled according to the user's settings rests on the correctness and security properties of the SeKernel. In designing and implementing both AIO and SeCloak, we strived to expose a narrow interface, minimize the TCB, and take care in writing and testing the implementation. Both of the enforcement layers are able to provide their guarantees even if the rest of the platform software

is compromised. However, we can still do better to strengthen the foundations of trust in these critical components.

In recent years, there have been many advances in practical formal methods that can be used to prove correctness and security properties for larger systems [58, 98, 116]. It would useful to formally verify parts of (or the entire) design and implementation of these enforcement layers to provide a much higher-level of assurance.

6.1.2 Usable Security

One of the goals for both AIO and SeCloak is to enable users to have fine-grained control over their devices. In SeCloak, we provide users the ability to reliably dictate on/off control for peripherals on their device, such as the microphone and camera. We present this control to users in the form of secure, virtual switches that they can interact with through a typical settings menu. We also provide certain presets that configure settings for given scenarios, such as airplane mode which reliably disables all radios (e.g., Bluetooth, WiFi, Cellular). While this interface makes sense to us as developers, it is worth determining whether it is most beneficial for lay users. Additionally, we make use of an LED to unambiguously notify the user that they are interacting with the secure part of SeCloak, but understanding the efficacy of this approach would be useful in comparison to other techniques (e.g., background tiles of a secure image known only to the user).

In AIO, the question of usable security is even greater since accountable paths

provide users with far more expressive policy control than in SeCloak. In addition, users are tasked with assigning credentials to principals (and their software); these credentials involve setting the privilege level for the software, which AIO uses to safely compose the software within the I/O stack and to determine which policies should be enforced. While accountable paths provide an appropriate abstraction at the level of the AIO enforcement layer, we envision higher-level abstractions that can help close the gap in usable security.

6.1.3 Cooperative Enforcement Layers

In both SeCloak and AIO, I assumed minimal cooperation between the enforcement layer and the (untrusted) platform OS. However, when considering deploying these enforcement layers on contemporary devices, there are several scenarios that would benefit from more cooperation between these two systems.

In SeCloak, we modify the platform OS only to enable strongly-ordered memory accesses for all memory mapped regions of device registers. Note that this modification is not needed for correctness, but rather for maintaining stability and usability of the device. The SeKernel needs precise data abort faults in order to trap and handle the access faults from the non-secure world accessing protected devices; if these were to go unhandled, the system would crash. However, forcing strongly-ordered memory accesses does introduce overhead for I/O operations, and only needs to be enabled for the memory regions corresponding to devices that are disabled via hardware protections. With an (untrusted) module running in the non-

secure world OS, the SeKernel could notify the OS module when the user explicitly confirms policy settings that change the set of devices that are protected. At that point, the OS module could alter the memory mappings for the devices that became protected (and vice-versa), before returning control back to the SeKernel to configure the protections. The OS could issue a denial of service attack at this point, but such attacks are outside of our threat model; the OS could simply not submit policy requests to the SeKernel in the first place from the settings application.

Additionally, such cooperation could be used to invoke device power management operations prior to protections. However, unlike the case of configuring strongly-ordered memory accesses, these management operations require interacting with the hardware device. Given that the user has directed SeCloak to disable the device at this time, we need to make sure that these operations are safe to perform. This notion of safety requires a behavior specification over valid accesses that can be made as part of power management routines to suspend a device, which the SeKernel could then enforce. How best to express these specifications, or whether there is a better alternative, is an open challenge.

In AIO, we introduce a (untrusted) proxy module into the platform OS to expose support for accountable paths to the platform OS and applications running on top. AIO maintains its own scheduler, which takes full control over a CPU core and only returns to the platform OS once all AIO module threads are not RUNNING/RUNNABLE. During this time, all interrupts assigned to the platform OS are masked and cannot be delivered on the CPU core. For practical deployment of AIO, the scheduler should be changed to cooperate with the scheduler in the

platform OS, such that all platform OS threads are not considered strictly lower priority than AIO threads.

6.2 Final Thoughts

As more aspects of our lives are captured and handled by our personal devices, it is important to reflect on how we can give users control over their devices and data. In recent years, there have been significant improvements with respect to privacy and security controls made available by the platform software on the personal devices. The introduction of trusted execution environments (TEEs) on these devices has definitely been a driving force for many of these improvements, as they provide a way to reduce the attack surface for critical components of the software platform through strong, hardware-enabled isolation. TEEs are used to provide a variety of secure functionality, such as managing biometric authentication information (e.g., fingerprints). However, the device manufacturer has complete control over the TEE software base. This is especially problematic due to the fact that the software base running in the TEE operates at the most-privileged level on the device (e.g., secure world OS in ARM TrustZone). Even if the user can control the OS and other platform software, such control may be subverted by the TEE software.

In this dissertation, I have discussed my work to provide users with control over their personal devices, while still enabling other principals, such as the device manufacturer, to implement and reason about their software stack. A core tenet of the enforcement layer is that the user has full privileges over their device. In

SeCloak, the user can directly express their on/off control policy for peripherals (e.g., camera, microphone) on their personal devices, which SeCloak will reliably enforce. In AIO, we enable users to apply more expressive policies, while also allowing users to delegate rights to other principals (e.g., device manufacturer, applications) to apply policies; AIO safely composes these policies, as well as software that implements functionality in the I/O stack, via the credential system. Going forward, I believe it is critical that we design our secure systems in a manner that provides users with control instead of unnecessarily locking their personal devices.

Bibliography

[1] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. SeCloak: ARM TrustZone-based Mobile Peripheral Control. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[2] Matthew Lentz, Peter Druschel, and Bobby Bhattacharjee. AIO: Control and Assurance Over Sensitive I/O Data. Under Submission, 2020.

[3] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. ACCessory: Password Inference Using Accelerometers on Smartphones. In *International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2012.

[4] Mariella Moon. Your phone's motion sensors can give away PINs and passwords. `https://www.engadget.com/2017-04-12-phone-motion-sensor-pin-vulnerability.html`, 2017.

[5] Patrick Lucas Austin. How to (Really) Turn Off Wi-Fi and Bluetooth in iOS 11. `https://lifehacker.com/how-to-really-turn-off-wi-fi-and-bluetooth-in-ios-11-1818655697`.

[6] Yulong Zhang, Zhaofeng Chen, Hui Xue, and Tao Wei. Fingerprints On Mobile Devices: Abusing and Leaking. `https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Fingerprints-On-Mobile-Devices-Abusing-And-Leaking-wp.pdf`.

[7] Jannis Müthing, Thomas Jäschke, and Christoph M Friedrich. Client-Focused Security Assessment of mHealth Apps and Recommended Practices to Prevent or Mitigate Transport Security Issues. *JMIR mHealth and uHealth*, 5(10), 2017.

[8] Dongjing He, Muhammad Naveed, Carl A Gunter, and Klara Nahrstedt. Security Concerns in Android mHealth Apps. In *AMIA Annual Symposium*, 2014.

[9] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. *International Conference on Trust and Trustworthy Computing (TRUST)*, 12, 2012.

[10] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2011.

[11] Viral Gandhi. Are Mobile Apps a Leaky Tap in the Enterprise? `https://www.zscaler.com/blogs/research/are-mobile-apps-leaky-tap-enterprise`.

[12] Charles Orton-Jones. Beware Household Gadgets That Can Take Control and 'Spy' on You. `https://www.raconteur.net/technology/beware-household-gadgets-that-can-take-control-and-spy-on-you`.

[13] Anthony Cuthbertson. Are Hackers Spying on Your Baby? `http://www.newsweek.com/search-engine-sleeping-babies-420967`.

[14] Matthew Brocker and Stephen Checkoway. iSeeYou: Disabling the MacBook Webcam Indicator LED. In *USENIX Security*, 2014.

[15] Rebecca S Portnoff, Linda N Lee, Serge Egelman, Pratyush Mishra, Derek Leung, and David Wagner. Somebody's Watching Me?: Assessing the Effectiveness of Webcam Indicator Lights. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2015.

[16] Trustonic. Trustonic. `https://www.trustonic.com/`.

[17] Qualcomm. Qualcomm Mobile Security. `https://www.qualcomm.com/products/features/security/mobile-security`.

[18] Google. Trusty TEE — Android Open Source Project. `https://source.android.com/security/trusty`.

[19] Samsung. Samsung KNOX. `https://www.samsungknox.com/en`.

[20] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *ACM Asia-Pacific Workshop on Systems (APSys)*, 2014.

[21] Ardalan Amiri Sani. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[22] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software Abstractions for Trusted Sensors. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[23] Stefan Saroiu and Alec Wolman. I am a sensor, and I approve this message. In *International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2010.

[24] Peter Gilbert, Jaeyeon Jung, Kyungmin Lee, Henry Qin, Daniel Sharkey, Anmol Sheth, and Landon P Cox. Youprove: Authenticity and Fidelity in Mobile Sensing. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

[25] Peter Gilbert, Landon P Cox, Jaeyeon Jung, and David Wetherall. Toward Trustworthy Mobile Sensing. In *International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2010.

[26] Yeongjin Jang, Simon P Chung, Bryan D Payne, and Wenke Lee. Gyrus: A Framework for User-Intent Monitoring of Text-based Networked Applications. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[27] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. Mimesis Aegis: A Mimicry Privacy Shield–A System's Approach to Data Privacy on Public Cloud. In *USENIX Security*, 2014.

[28] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. VButton: Practical Attestation of User-Driven Operations in Mobile Apps. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[29] Kailiang Ying, Priyank Thavai, and Wenliang Du. Truz-View: Developing TrustZone User Interface for Mobile OS using Delegation Integration Model. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2019.

[30] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. Establishing Trusted I/O Paths for SGX Client Systems with Aurora. *IEEE Transactions on Information Forensics and Security (TIFS)*, 2019.

[31] Anish Athalye, Adam Belay, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A Device for Secure Transaction Approval. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[32] Zongwei Zhou, Virgil D Gligor, James Newsome, and Jonathan M McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *IEEE Symposium on Security and Privacy*, 2012.

[33] Zongwei Zhou, Miao Yu, and Virgil D Gligor. Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O. In *IEEE Symposium on Security and Privacy*, 2014.

[34] Yueqiang Cheng, Xuhua Ding, and Robert H Deng. DriverGuard: Virtualization-based Fine-grained Protection on I/O Flows. *ACM Transactions on Information and System Security (TISSEC)*, 16(2), 2013.

[35] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[36] Intel. Intel Virtualization Technology (Intel VT). `https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html`.

[37] AMD. Virtualization Solutions — AMD. `https://www.amd.com/en/technologies/virtualization-solutions`.

[38] ARM. Virtualization is Coming to a System Near You. `https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`.

[39] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ACM SIGARCH Computer Architecture News*, volume 36, 2008.

[40] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure Applications on an Untrusted Operating System. In *ACM SIGARCH Computer Architecture News*, volume 41, 2013.

[41] Youngjin Kwon, Alan M Dunn, Michael Z Lee, Owen S Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *ACM SIGPLAN Notices*, volume 51, 2016.

[42] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[43] Intel. Intel SGX Homepage. `https://software.intel.com/en-us/sgx`.

[44] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.

[45] ARM. ARM Trustzone. `https://www.arm.com/products/security-on-arm/trustzone`.

[46] Intel. Intel Trusted Execution Technology. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf`.

[47] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://support.amd.com/TechDocs/24593.pdf.

[48] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *European Conference on Computer Systems (EuroSys)*, 2008.

[49] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), 2015.

[50] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[51] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[52] ARM. ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition. https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf, 2014.

[53] ARM. ARM Architecture Reference Manual ARMv8, for ARMv8-A Architecture Profile. https://static.docs.arm.com/ddi0487/fb/DDI0487F_b_armv8_arm.pdf, 2020.

[54] ARM. SMC Calling Convention. http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf.

[55] ARM. ARM CoreLink TZC-400 TrustZone Address Space Controller. https://static.docs.arm.com/100325/0001/arm_corelink_tzc400_trustzone_address_space_controller_trm_100325_0001_02_en.pdf, 2015.

[56] ARM. ARM Generic Interrupt Controller Architecture Specification. https://static.docs.arm.com/ihi0069/c/IHI0069C_gic_architecture_specification.pdf, 2016.

[57] ARM. ARM Security Technology Building a Secure System using TrustZone Technology. https://static.docs.arm.com/genc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2020.

[58] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[59] Yeongpil Cho, Jun-Bum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *USENIX Annual Technical Conference*, 2016.

[60] Linaro. OP-TEE Home. `https://www.op-tee.org/`.

[61] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security*, 2011.

[62] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard - Enforcing User Requirements on Android Apps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[63] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[64] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[65] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[66] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing Android Permission Creep. In *Web 2.0 Security and Privacy (W2SP)*, 2011.

[67] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming Information-stealing Smartphone Applications (on Android). In *International Conference on Trust and Trustworthy Computing (TRUST)*, 2011.

[68] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *USENIX Security*, 2015.

[69] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *IEEE International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[70] Sebastian Neuner, Victor Van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter Sandbox: Android Sandbox Comparison. *arXiv preprint arXiv:1410.7749*, 2014.

[71] Ayush Sahu and Ashima Singh. Securing IoT devices using JavaScript based Sandbox. In *Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, 2016.

[72] Paul A Karger, Mary Ellen Zurko, Douglas W Bonin, Andrew H Mason, and Clifford E Kahn. A VMM Security Kernel for the VAX Architecture. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1990.

[73] Stanley R Ames Jr, Morrie Gasser, and Roger R Schell. Security Kernel Design and Implementation: An Introduction. *IEEE Computer*, 16(7), 1983.

[74] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A Security Enforcement Kernel for OpenFlow Networks. In *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2012.

[75] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security*, 2002.

[76] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[77] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-Related Policy Enforcement for Android. In *International Conference on Information Security (ISC)*, volume 10, 2010.

[78] David Zeuthen. polkit: polkit Reference Manual. `https://www.freedesktop.org/software/polkit/docs/latest/polkit.8.html`.

[79] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference*, 2001.

[80] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating ARM TrustZone Devices in Restricted Spaces. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[81] Nuno Santos, Nuno O Duarte, Miguel B Costa, and Paulo Ferreira. A Case for Enforcing App-Specific Constraints to Mobile Devices by Using Trust Leases. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[82] Miguel B Costa, Nuno O Duarte, Nuno Santos, and Paulo Ferreira. TrUbi: A System for Dynamically Constraining Mobile Devices within Restrictive Usage Scenarios. In *ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2017.

[83] Saeed Mirzamohammadi, Justin A Chen, Ardalan Amiri Sani, Sharad Mehrotra, and Gene Tsudik. Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2017.

[84] Saeed Mirzamohammadi and Ardalan Amiri Sani. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[85] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, et al. Bitvisor: A Thin Hypervisor for Enforcing I/O Device Security. In *ACM International Conference on Virtual Execution Environments (VEE)*, 2009.

[86] Landon P Cox and Peter M Chen. Pocket Hypervisors: Opportunities and Challenges. In *IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2007.

[87] Alan M Dunn, Michael Z Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[88] Archana Ganapathi, Viji Ganapathi, and David A Patterson. Windows XP Kernel Crash Analysis. In *Large Installation System Administration Conference (LISA)*, 2006.

[89] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *USENIX Annual Technical Conference*, 2010.

[90] Microsoft. Overview of UMDF. `https://docs.microsoft.com/en-us/windows-hardware/drivers/wdf/overview-of-the-umdf`, 2017.

[91] Hans-J Koch. The Userspace I/O HOWTO – The Linux Kernel Documentation. `https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html`, 2006.

[92] Simon Biggs, Damon Lee, and Gernot Heiser. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In *ACM Asia-Pacific Workshop on Systems (APSys)*, 2018.

[93] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. *ACM SIGOPS Operating Systems Review*, 29(5), 1995.

[94] Christopher A Small and Margo I Seltzer. Vino: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard Computer Science Group, 1994.

[95] Linux. BPF Documentation. `https://www.kernel.org/doc/html/latest/bpf/index.html`.

[96] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Technical Conference*, 1993.

[97] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[98] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[99] Google. Zircon. `https://fuchsia.dev/fuchsia-src/concepts/kernel`, 2020.

[100] Google. Fuchsia. `https://fuchsia.dev/`, 2020.

[101] David Mosberger and Larry L Peterson. Making Paths Explicit in the Scout Operating System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[102] Oliver Spatscheck and Larry L Peterson. Escort: A Path-based OS Security Architecture. Technical Report TR 97-17, University of Arizona, Tucson, AZ, 1997.

[103] devicetree.org. Devicetree Specification Release 0.1. `https://www.devicetree.org/downloads/devicetree-specification-v0.1-20160524.pdf`.

[104] Renju Liu and Mani Srivastava. PROTC: PROTeCting Drone's Peripherals through ARM TrustZone. In *Workshop on Micro Aerial Vehicle Networks, Systems, and Applications (DroNet)*, 2017.

[105] DENX Software Engineering. Das U-Boot – The Universal Boot Loader. `https://www.denx.de/wiki/U-Boot`.

[106] NXP. i.MX 6 Series Applications Processors. https://www.nxp.com/products/microcontrollers-and-processors/arm-based-processors-and-mcus/i.mx-applications-processors/i.mx-6-processors:IMX6X_SERIES.

[107] ARM. Cortex-A Series Programmer's Guide, Version: 4.0. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0013d/index.html.

[108] ARM. ARM Power State Coordination Interface. http://infocenter.arm.com/help/topic/com.arm.doc.den0022d/Power_State_Coordination_Interface_PDD_v1_1_DEN0022D.pdf.

[109] Zoltan Pallagi. WiFi Speed Test - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.pzolee.android.localwifispeedtester.

[110] Thorsten Schreiber. Android Binder – Android Interprocess Communication. Master's thesis, Ruhr University Bochum, 10 2011.

[111] Franco Travostino, Ed Menze, and Franklin Reynolds. Paths: Programming with System Resources in Support of Real-time Distributed Applications. In *Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, 1996.

[112] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short Group Signatures. In *International Cryptology Conference (CRYPTO)*, 2004.

[113] Paul England, Andrey Marochko, Dennis Mattoon, Rob Spiger, Stefan Thom, and David Wooten. RIoT - A Foundation for Trust in the Internet of Things. Technical Report MSR-TR-2016-18, 2016.

[114] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security*, 2017.

[115] ARM. SSL Library mbed TLS /Polar SSL. https://tls.mbed.org/.

[116] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-performance Cryptographic Assembly Code. In *USENIX Security*, 2017.