UNIVERSITY OF CALIFORNIA,
IRVINE

Security Monitor for Mobile Devices: Design and Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Saeed Mirzamohammadi

2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to express my gratitude to my advisor Ardalan Amiri Sani. Throughout these years, his guidance, enthusiasm, and passion was the main driver for me to become a better researcher.. Without his knowledge and guidance, it was impossible for me to finish my PhD and I learned a lot from him. He is a wonderful mentor and I am very thankful for his patience toward me.

I am also thankful to my committee members, Gene Tsudik, Sharad Mehrotra, and Sharad Agarwal. Their continuous support and feedback on this dissertation have been of great value.

I am also grateful to my group mates at TrussLab including Seyed Mohammadjavad Seyed Talebi, Myles Liu, and Zhihao Yao. I am deeply thankful for my amazing family and friends that have always have been supportive of me during the course of my PhD.

I would like to acknowledge the funding support made by the National Science Foundation (NSF) Award #1617513 and #1718923.

VITA

Saeed Mirzamohammadi

EDUCATION

Ph.D. Candidate in Computer Science	**2020**
University of California, Irvine	*Irvine, CA*
M.Sc. in Computer Science	**2019**
University of California, Irvine	*Irvine, CA*
B.Sc. in Computer Engineering	**2014**
Sharif University of Technology	*Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2015–2020**
University of California, Irvine	*Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant	**2014–2015, 2020**
University of California, Irvine	*Irvine, CA*

REFEREED JOURNAL PUBLICATIONS

Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems	**2018**
IEEE Transactions on Mobile Computing (TMC)	

REFEREED CONFERENCE PUBLICATIONS

Tabellion: Secure Legal Contracts on Mobile Devices	**Jun 2020**
Proc. ACM Int. Conf. Mobile Systems, Applications and Services (MobiSys)	
Milkomeda: Safeguarding the Mobile GPU Interface Using WebGL Security Checks	**Oct 2018**
Proc. ACM Conference on Computer and Communications Security (CCS)	
The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices	**Aug 2018**
Proc. ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)	

Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices **Nov 2017**
Proc. ACM Int. Conf. Embedded Networked Sensor Systems (SenSys)

Understanding Sensor Notifications on Mobile Devices **Feb 2017**
Proc. ACM Int. Workshop on Mobile Computing Systems and Applications (HotMobile)

Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems **Jun 2016**
Proc. ACM Int. Conf. Mobile Systems, Applications and Services (MobiSys)

SOFTWARE

Passive hypervisor and its benchmarks `https://trusslab.github.io/hyp_tee/`
APSys'18

Ditio `https://trusslab.github.io/ditio/`
SenSys'17

Notification user study data `https://www.ics.uci.edu/~ardalan/notifdata.html/`
HotMobile'17

Viola `https://trusslab.github.io/viola/`
MobiSys'16

ABSTRACT OF THE DISSERTATION

Security Monitor for Mobile Devices: Design and Applications

By

Saeed Mirzamohammadi

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Ardalan Amiri Sani, UCI, Chair

Android's underlying Linux kernel is rapidly becoming an attractive target for attackers. In 2014, the number of reported bugs in the kernel was 4 percent of the overall bugs discovered in Android. This number drastically increased to 9 and 44 percent in 2015 and 2016, respectively. These are not surprising as the kernel consists of millions of lines of code, which bloat the Trusted Computing Base (TCB) and enlarge the attack surface. For example, several recent bugs have been found in the Bluetooth [36] and WiFi [37, 172] subsystems of mobile devices. These bugs can be exploited by malicious parties over the network to mount remote attacks. An attacker uses these kernel bugs to get kernel privilege and gain complete control of the mobile device.

The common solution to this problem is to patch the kernel as soon as a new bug or a vulnerability is found in the kernel. However, this does not protect the system against zero-day vulnerabilities. In this dissertation, we present the *security monitor*, a small, trustworthy, and extensible software that provides different security services, with a small TCB. The security services within the security monitor enforce certain privacy and security guarantees for the system, e.g., enforcing certain privacy guarantees for the use of I/O devices. With the security monitor, a compromised operating system will not be able to undermine these guarantees. The security monitor is designed and built based on ARM TrustZone [24, 5, 26, 176] and virtualization hardware [70, 72], which are available in modern mobile and IoT devices. The hypervisor layer, supported by virtualization hardware, enables the security monitor to efficiently control accesses to certain part of the physical memory, e.g., registers of selected sensors, without making any modifications to the operating system. TrustZone, which is already being used by many security applications by vendors, provides a complementary use for the security monitor. It enables sealing of data to guarantee integrity, authenticity, and confidentiality.

One concern about the security monitor is performance. It is widely believed that running the main operating system on top of a hypervisor incurs significant performance overhead [93]. Through extensive experimentation and a redesign of the hypervisor, we provide evidence against this argument. We show that a commodity hypervisor's overhead is mainly due to its frequent interposing on the operating system activities, a design needed only in a multi-tenant virtualization setup. The hypervisor can be redesigned to minimize these interpositions and hence minimize its performance overhead on the main OS. We show that the performance of the operating system is very close to that of native execution.

We demonstrate three end-to-end systems that leverage the security monitor to design different security services: *(i)* trustworthy sensor notifications using low-level checks in the security monitor (Viola), *(ii)* trustworthy auditing of sensor activities by recording the sensor activities in the security monitor (Ditio), and *(iii)* secure formation of electronic contracts by designing secure primitives in the security monitor (Tabellion).

Chapter 1

Introduction

Android's underlying Linux kernel is rapidly becoming an attractive target for attackers. In 2014, the number of reported bugs in the kernel was 4 percent of the overall bugs discovered in Android. This number drastically increased to 9 and 44 percent in 2015 and 2016, respectively. These are not surprising as the kernel consists of millions of lines of code, which bloat the Trusted Computing Base (TCB) and enlarge the attack surface. For example, several recent bugs have been found in the Bluetooth [36] and WiFi [37, 172] subsystems of mobile devices. These bugs can be exploited by malicious parties over the network to mount remote attacks. An attacker uses these kernel bugs to get kernel privilege and gain complete control of the mobile device.

The common solution to this problem is to patch the kernel as soon as a new bug or a vulnerability is found in the kernel. However, this does not protect the system against zero-day vulnerabilities. In this dissertation, we present the *security monitor*, a small, trustworthy, and extensible software that provides different security services, with a small TCB. The security services within the security monitor enforce certain privacy and security guarantees for the system, e.g., enforcing certain privacy guarantees for the use of I/O devices. With the security monitor, a compromised operating system will not be able to undermine these guarantees. The security monitor is designed and built based on ARM TrustZone [24, 5, 26, 176] and virtualization hardware [70, 72], which are available in modern mobile and IoT devices. The hypervisor layer, supported by virtualization hardware, enables the security monitor to efficiently control accesses to certain part of the physical memory, e.g., registers of selected sensors, without making any modifications to the operating system. TrustZone, which is already being used by many security applications by vendors, provides a complementary use for the security monitor. It enables sealing of data to guarantee integrity, authenticity, and confidentiality. We will discuss the design of the security monitor in chapter §2.

We design the security monitor with practicality in mind. Its design is low-level and on-demand. Its low-level design allows it to be deployed and locked by the mobile and IoT device vendors (using the secure boot feature [24]) without modifications to the operating system. Its generic design allows the security monitor to be easily ported to different mobile and IoT devices with diverse hardware configurations. Its on-demand nature means that these services can be turned on and off as needed. Therefore, the security monitor's runtime overhead, although not significant, can be fully avoided when the service is not needed.

One concern about the security monitor is performance. It is widely believed that running the main operating system on top of a hypervisor incurs significant performance overhead [93]. Through extensive experimentation and a redesign of the hypervisor, we provide evidence against this argument. We show that a commodity hypervisor's overhead is mainly due to its frequent interposing on the operating system activities, a design needed only in a multi-tenant virtualization setup. The hypervisor can be redesigned to minimize these interpositions and hence minimize its performance overhead on the main OS. In chapter §3, we show that the performance of the OS is very close to that of native execution.

In §2.2, we also propose a version of the security monitor that is only based on the virtualization hardware. In this proposal, we suggest that the security services that are already implemented by TrustZone can be implemented based on the virtualization hardware. Virtualization hardware can be used to implement multiple isolated trusted environments, as opposed to a single such environment provided by TrustZone. This can prevent the bloat of the TCB of the trusted execution environment (TEE) and support new security services not currently possible, such as sandboxing of untrusted operating system kernel components. We will discuss and address the concerns on supported features, backward-compatibility, and hypervisor's TCB size.

We have developed three end-to-end systems that leverage the security monitor to design different security services. We will introduce these systems in the rest of this chapter and will then discuss them in detail in chapters §4, §5, and §6.

1.1 Applications of the security monitor

1.1.1 Trustworthy Sensor Notifications

Modern mobile systems such as smartphones, tablets, and wearables contain a plethora of sensors such as camera, microphone, GPS, and accelerometer. Moreover, being mobile, these systems are with the user all the time, e.g., in user's purse or pocket. Therefore, mobile sensors can capture extremely sensitive and private information about the user including daily conversations, photos, videos, and visited locations. Such a powerful sensing capability raises important privacy concerns.

To address these concerns, we believe that mobile systems must be equipped with trustworthy sensor notifications, which use indicators such as LED to inform the user unconditionally when the sensors are on. In §4, we present Viola, our design and implementation of trustworthy sensor notifications using the security monitor. The security monitor intercepts writes to the registers of sensors and indicators, evaluates them against checks on sensor notification invariants, and rejects those that fail the checks. We also use formal verification methods to prove the functional correctness of the compilation of our invariant checks from a high-level language. We demonstrate the effectiveness of Viola on different mobile systems, such as Nexus 5, Galaxy Nexus, and ODROID XU4, and for various sensors and indicators, such as camera, microphone, LED, and vibrator. We demonstrate that Viola incurs almost no overhead to the sensor's performance and incurs only small power consumption overhead.

1.1.2 Trustworthy Sensor Auditing

Mobile and Internet-of-Things (IoT) devices, such as smartphones, tablets, wearables, smart home assistants (e.g., Google Home and Amazon Echo), and wall-mounted cameras, come equipped with various sensors, notably camera and microphone. These sensors can capture extremely sensitive and private information. There are several important scenarios where, for privacy reasons, a user might require assurance about the use (or non-use) of these sensors. For example, the owner of a home assistant might require assurance that the microphone on the device is not used during a given time of the day. Similarly, during a confidential meeting, the host needs assurance that attendees do not record any audio or video. Currently, there are no means to attain such assurance in modern mobile and IoT devices.

To this end, in §5, we present Ditio, a system approach for *auditing sensor activities.* Ditio records sensor activity logs that can be later inspected by an auditor and checked for compliance with a given policy. Ditio recorder is built based in the security monitor. Ditio includes an authentication protocol for establishing a logging session with a trusted server and a formally verified companion tool for log analysis. Ditio prototypes on ARM Juno development board and Nexus 5 smartphone show that it introduces negligible performance overhead for both the camera and microphone. However, it incurs up to 17% additional power consumption under heavy use for the Nexus 5 camera.

1.1.3 Secure Legal Contracts

A legal contract is an agreement between two or more parties as to something that is to be done in the future. Forming contracts electronically is desirable since it is convenient. However, existing electronic contract platforms have a critical shortcoming. They do not provide strong evidence that a contract has been legally and validly created. More specifically, they do not provide strong evidence that an electronic signature is authentic, that there was mutual assent, and that the parties had an opportunity to read the contract.

In §6, we present Tabellion, a system for forming legal contracts on mobile devices, such as smartphones and tablets, that addresses the above shortcoming. We define four secure primitives and use them in Tabellion to introduce self-evident contracts, the validity of which can be verified by independent inspectors. We show how these primitives can be implemented securely in the security monitor as well as a secure enclave in a centralized server, all with a small TCB. Moreover, we demonstrate that it is feasible to build a fully functional contract platform on top of these primitives. We develop ∼15000 lines of code (LoC) for our prototype, only ∼1000 of which need to be trusted. Through analysis, prototype measurements, and a 30-person user study, we show that Tabellion is secure, achieves acceptable performance, and provides better usability than the state-of-the-art electronic contract platform, DocuSign, for viewing and signing contracts. to demonstrate correct and legal formation of a contract, which, according to the law of contracts, requires demonstration of signature attribution, mutual assent, and reading opportunity.

Chapter 2

Security monitor Design

In this chapter, we elaborate on the design of the security monitor using virtualization hardware and ARM TrustZone. We also propose a virtualization-based TEE, in which we suggest using only the virtualization for implementing TEE rather than TrustZone.

2.1 Design Overview

Security monitor is based on two important hardware features in modern ARM processors. **The first** is ARM TrustZone [5, 26, 176], which is a system-wide hardware isolation feature for ARM SoCs. It splits execution into two worlds: *normal world* hosting the main operating system and *secure world* hosting a secure runtime as well as a secure monitor to handle switching between these two worlds. The secure world hosts the TEE, which is a secure operating system with its own user and kernel spaces. Some popular OSes used in the secure world are OP-TEE [40] and Trusty OS [34]. TrustZone partitions the memory and hardware I/O devices between the normal and secure worlds. All accesses from the normal world to the resources allocated for the secure world are denied. The secure world, however, has access to all resources. The TEE is used to host security services, such as payment services. These services can be invoked from the normal world through a regulated call gate, enabled by a world switch instruction called the "Secure Mode Call" (SMC).

The second is hardware support for virtualization, which creates a new privilege level in the normal world, called the hyp mode [97], in addition to existing kernel and user modes that is entered from the kernel by a new instruction called the "Hypervisor Call" (HVC). Virtualization hardware was added to ARM processors in 2011 [146]. However, due to lack of critical use cases, this hardware is typically deactivated in commodity mobile devices.

These hardware features are available on many new ARM processors. Examples are 32-bit processors, such as Cortex A7 and Cortex A15 [70], and 64-bit processors, such as Cortex A53 and Cortex A57 [72], which are used in most modern smartphones and tablets, such as Nexus 6P smartphone and Pixel C tablet (both using 4 Cortex A57 cores and 4 Cortex A53 cores in the big.LITTLE architecture). Moreover, these processors are used in modern high-end IoT devices and wearables as well. For example, Google Home leverages a dual-core ARM Cortex A7 processor [9]. Similarly, Samsung Gear 2 smartwatch leverages a dual-core ARM Cortex A7 processor in its Exynos 3250 SoC [19].

The main benefit of our security monitor is its usage of complementary capabilities of both hardware features. Virtualization extension enables the security monitor to control accesses to the main memory isolated from the main operating system using stage-2 page tables, and TrustZone enables the security monitor to seal any data using cryptographic libraries and keys that are only accessible in the secure world. Figure 2.1 overviews the security monitor. Communication between the hypervisor and secure world is attained through a protected channel implemented using shared memory and Secure Monitor Call (SMC) [24], which performs a context switch between normal and secure worlds.

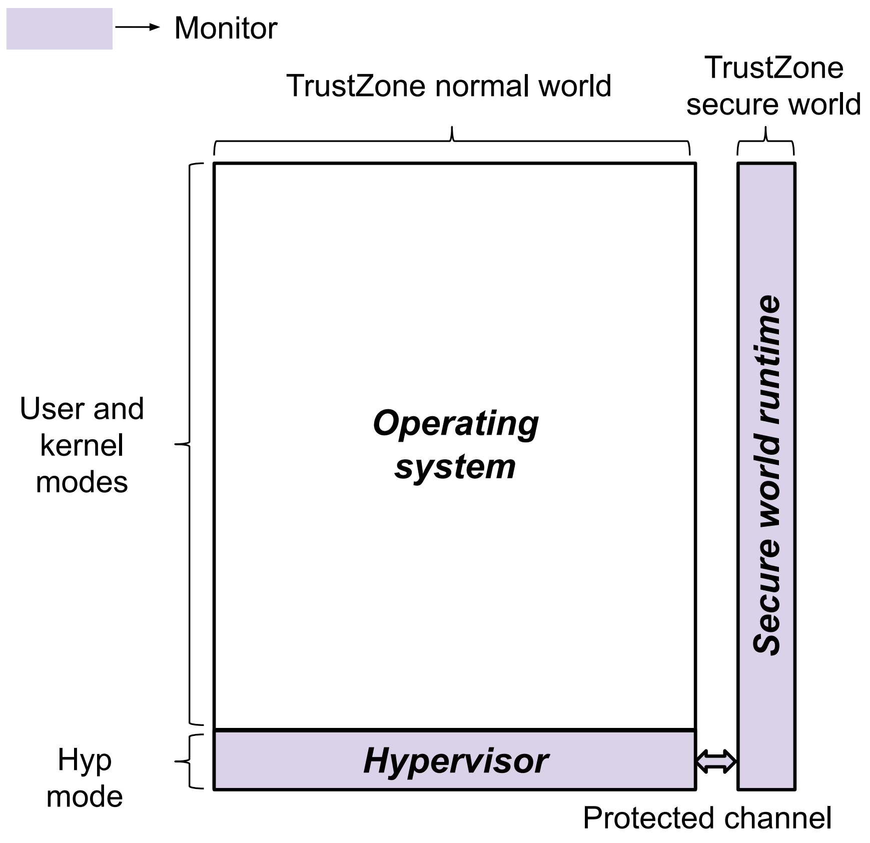


Figure 2.1: Security monitor overview.

Threat model: We have designed the security monitor to enforce the security services and protect these services from a malicious user or operating system. Security monitor protects the services because the security services run in the security monitor and the security monitor is isolated from the operating system by virtualization and TrustZone hardware. We assume that the security monitor is not compromised. We assume that an attacker can access the victim's device (e.g., by stealing it) but cannot compromise the security monitor. A remote attacker can install a malicious application on the victim's device and take full control of the device. Security monitor can use a hardware-bound private key and a device certificate that is only accessible by the security monitor to prove its identity to a remote entity and provide authenticity and integrity of data, if needed. We trust the hardware of the mobile devices, however, we do not trust the drivers interacting with them in the operating system.

Security monitor does not protect against side channel attacks that target the hypervisor or TrustZone secure world such as cache side channel attacks [131]. Security monitor does not protect against Denial-of-Service (DoS) attacks. First, an attacker that has control over the operating system may deny forwarding any requests to the security services or deny forwarding any received data from the security services in the security monitor to the user or a remote party. Second, an attacker can take advantage of the bugs in the security monitor to perform DOS attacks.

Invoking the security monitor: The security monitor provides a hypercall and SMC call for the normal world operating system to request a service. The hypercall and SMC call may need to pass a few additional data as well, depending on the security service. For instance, it might need to pass the physical address of an I/O device registers or a buffer that is allocated in the operating system in the driver. The security service in the security monitor does not trust the data it receives from the untrusted operating system. The security service might perform further sanity checks on the data or cryptographically sign the data for future audits to make sure that it is properly validated.

Controlling Accesses to Memory: The security monitor can intercept all accesses to the memory using ARM's stage-2 page tables. To intercept accesses, the hypervisor removes the Stage-2 page table entry's read and write permissions to intercept the operating system's accesses to a particular page. This enables the security monitor to intercept all accesses from the operating system, e.g., sensor register accesses, using ARM's Stage-2 page tables. This forces the accesses to trap into the hypervisor. The security monitor can then inspect the content of the CPU registers in the trap handler and decode the trapped instruction to read the access parameters. It can then pass the parameters to the security service. The security service will then decide how to proceed with the access, e.g., record or emulate the access, before returning from the trap or reject the access completely. Emulation is done by reading from or writing to the same physical address through a secondary mapping in the hypervisor.

Backward-compatibility: This issue comes up since: (*i*) older mobile and low-end IoT devices might not be equipped with virtualization hardware and/or TrustZone, and (*ii*) even if they are, both virtualization hardware and TrustZone are not programmable by non-vendors on commodity devices. Therefore, to expand the applicability of security monitor, we provide a backward-compatible design and implementation that uses the operating system kernel instead of the hypervisor and TrustZone secure world. The main disadvantage of this backward-compatible design is increased TCB size, since the operating system kernel needs to be trusted.

The backward-compatible design enables tech-savvy users to deploy security monitor on their devices. However, in practice, we expect security monitor to be deployed by mobile and IoT vendors, who can deploy the this design on their new devices and deploy the backward-compatible design only on older devices that do not meet the hardware requirements of the security monitor.

2.2 Virtualization-based security monitor

In this section, we propose an alternative design of the security monitor that is only based on the virtualization hardware. In this design, we propose that the security services that are already implemented by TrustZone can be implemented using the virtualization hardware. Virtualization hardware can be used to implement multiple isolated trusted environments, as opposed to a single such environment provided by TrustZone. This can prevent the bloat of the TCB of the security monitor and support new security services not currently possible, such as sandboxing of untrusted operating system kernel components, compared to the design of current systems that is only based on the TrustZone. We will discuss and address the concerns on supported features, backward-compatibility, and hypervisor's TCB size, in this chapter. It is important to note that we will not use this alternative design in the rest of this dissertation.

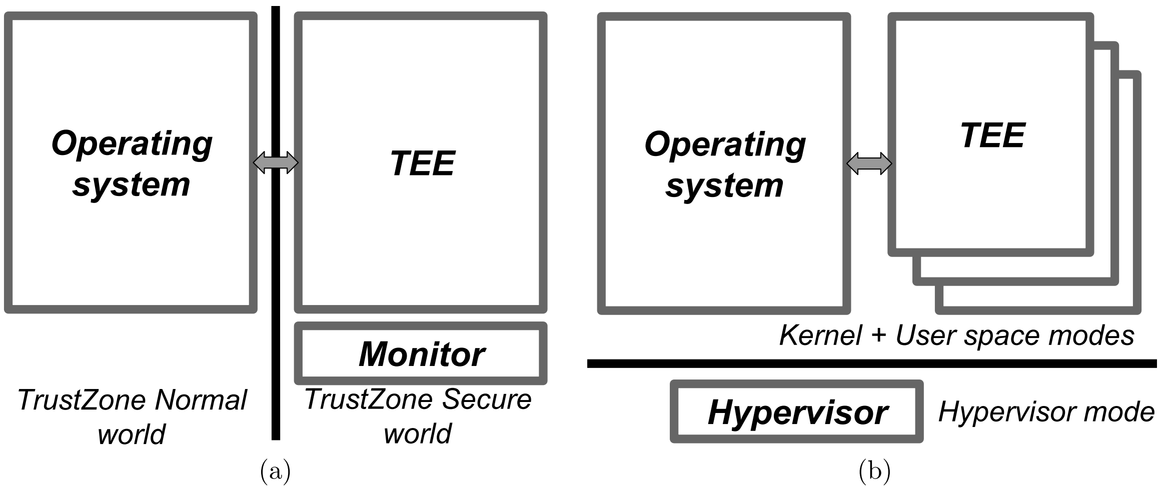


Figure 2.2: (a) TrustZone-based TEE architecture. (b) Virtualization-based TEE architecture.

In today's mobile devices, TrustZone is predominantly used to implement a TEE, which hosts important security services such as secure payment (e.g. Samsung Pay), Digital Rights Management (DRM), and a cryptographic key store. Virtualization hardware, on the other hand, is mainly intended for supporting virtual machines, and hence is typically left unused in mobile devices. In this section, we make the case for using virtualization hardware to implement the TEE.

Figure 2.2a shows the current design of a TEE architecture and Figure 2.2b illustrates how this hardware can be used to implement the TEE. The main operating system as well as the TEEs run inside virtual machines on top of a hypervisor. The hypervisor then gives each virtual machine access to the resources it needs. For example, it gives a TEE access to its own secure memory and secure I/O devices, while it gives the main operating system access to unsecure memory and I/O. The operating system and TEEs can also communicate through the hypervisor. The mediation of the hypervisor increases the round trip time for this communication compared to the direct communication with TrustZone. However, we do not anticipate this to be an issue since security services hosted in TEEs are typically not performance-sensitive.

As the figure shows, this design enables having multiple TEEs. This provides two important advantages compared to a single TEE supported by ARM TrustZone. First, it allows for strong isolation between various security services. Recently, many novel security services have been proposed by the research community to be deployed within the TrustZone TEE. Examples are secure sensors [132], AdAttester [127], VButton [128], and TruZ-Droid [178]. However, these services require deploying extra code (e.g., device drivers) in the TEE, mainly in the kernel of the TEE operating system. Such a requirement is an important bottleneck for the adoption of these solutions in practice. This is because doing so will noticeably increase the TCB of the TEE, which hosts critical security services. With the virtualization-based TEE design, these services can be deployed in separate TEEs, without bloating the TCB.

Second, this design allows for sandboxing of the operating system kernel's components. For example, several recent bugs have been found in the Bluetooth [36] and WiFi [37, 172] subsystems of mobile devices. These bugs can be exploited by malicious parties over the network to mount remote attacks. An effective solution to prevent such exploits is to sandbox these network devices and their corresponding device drivers. Unfortunately, TrustZone TEE cannot be used for this purpose. This is because the TEE houses critical security services and adding these vulnerable devices to this TEE makes it vulnerable to attacks, essentially worsening the security of the system. On the other hand, with a virtualization-based design, isolated virtual machines can be used to sandbox these network devices, similar to Cinch [67]. These virtual machines will be given access only to their own I/O devices, hence properly isolating the vulnerabilities.

While a virtualization-based TEE provides those advantages, it raises some concerns that prohibits its widespread use: performance, supported features, backward-compatibility, and hypervisor's TCB size. We address the concerns on performance in §3 and the supported features in the rest of this chapter. The concern on backward-compatibility is easily addressed by porting existing TrustZone TEE and applications to run in a virtual machine, similar to vTZ [117].

Here, we study the issue of TCB size. As discussed, a virtualization-based TEE allows for reduction in the TCB of the TEE software itself. However, there is concern with the large size of the hypervisor compared to the monitor code in the secure world (Figure 2.2). Our initial code size measurement shows that Xen hypervisor's TCB for ARM architecture is about 95 kLoC while TrustZone's monitor TCB (i.e., ARM Trusted Firmware) is about 28 kLoC. Our breakdown of Xen's TCB shows that this number can be significantly reduced. For example, about 36 kLoC is for drivers, 20 kLoC is for tools, and 2 kLoC is for cryptography, none of which is needed in our passive hypervisor. This leaves us about 36 kLoC for TCB. Indeed, others have also found that a hypervisor's TCB size can be reduced. For example, in [30], it is mentioned that the ARM code size in Xen can be reduced down to about 11 kLoC. Moreover, [168] and [164] also have shown that the hypervisor code size can be reduced to about 9 and 20 kLoC, respectively. Also, note that ARM Trusted Firmware (in the HiKey development board used in our prototype) can also be reduced to about 17 kLoC by removing its services and tools. This analysis shows that both Xen and ARM Trusted Firmware have comparable TCBs. Note that while we focus on a commodity hypervisor (Xen) in this paper, we could also use microkernels or microhypervisors such as [32, 114, 122, 168]. In the rest of this chapter, we address the concern on supported features for a virtualization-based TEE.

2.2.1 Secure Memory

TrustZone supports secure memory for the TEE. It does so by performing checks on the physical address requests sent to TZASC. Virtualization hardware can also support isolated secure memory for its TEE domains. It does so using a two-stage address translation, which allows it to isolate the memory pages assigned to different virtual machines.

Indeed, we argue that virtualization's method of implementing secure memory is superior. First, it allows the hypervisor to assign memory to different domains at a 2 MB super page granularity (a smaller page size is possible but it degrades performance as shown in §3.1.1). On the other hand, TrustZone can only create 8 regions [71] and each region must be allocated contiguously on the physical memory.

Second, with an effective TLB algorithm, virtualization's overhead on memory access can be made to be very small. However, TrustZone always incurs a constant overhead of 2 cycles on memory writes as mentioned in section §3.2. Since TrustZone performs the security checks on the physical address, TLB cannot reduce the overhead.

2.2.2 Secure I/O

TrustZone supports secure I/O for the TEE. That is, it can assign an I/O device to the TEE domain, in which case the I/O device will not be accessible to the normal world operating system. This is the technique behind several research efforts using ARM TrustZone including secure sensors [132], AdAttester [127], VButton [128], and TruZ-Droid [178].

Virtualization can also support secure I/O by assigning an I/O device to a virtual machine (a technique also known as direct device assignment [64, 133, 112, 79, 134]). In this technique, the registers of the device is mapped to the virtual machine's physical address space, the interrupts are redirected to the virtual machine, and the Direct Memory Access (DMA) operations of the I/O devices are limited to the virtual machine memory using I/O Memory Management Units (IOMMUs).

Indeed, we argue that virtualization's method of implementing secure I/O is superior. First, virtualization allows to secure I/O device interface partially. For example, Viola only monitors the operating system accesses to some registers of an I/O device [147, 149]. And SchrodinText only controls the GPU and display's access to the framebuffer using IOMMUs [65]. Implementing such systems with TrustZone's secure I/O requires giving full control of these I/O devices to the TEE, which unnecessarily bloats its TCB.

Second, the performance of I/O devices are also affected by the security checks performed by the TZASC. More specifically, CPU write access to I/O device registers as well as DMA writes are subject to the 2 cycle overhead of TZASC (and reads can suffer from overhead too if the limited number of speculative accesses are not adequate as mentioned in §3.2). Similar to memory access, virtualization also adds overhead to these operations due to address translations. However, virtualization's overhead can be mitigated by a good translation caching algorithm.

2.2.3 Secure Boot & Cryptographic Keys

TrustZone's secure boot allows it to check the integrity of the operating system image before loading it. Moreover, the cryptographic keys accessible only in the secure world allows for implementation of various cryptographic protocols (including the integrity check used in secure boot) as well as a key store used by applications. These features are not currently supported in virtualization hardware. However, we argue that they can simply be added. First, the cryptographic keys are burned on some form of a read-only memory (e.g., "One Time Programmable (OTP) or eFuse memory" [73]) only available to the secure world (the EL3 privilege level). To make them available to the hypervisor, they should be simply be made accessible to the EL2 privilege level. We believe that hardware modification needed to achieve this is trivial.

Second, secure boot is implemented in multiple stages in software (e.g., in the BIOS, bootloader, and the secure world code) [160]. Therefore, adding this feature to the hypervisor only requires modifications to these software layers.

Chapter 3

Performance of the security monitor

In this chapter, we study the performance of the security monitor. Using the security monitor requires the hypervisor in the system all the time. This, in turn, can affect the performance of the main operating system. In this chapter, we investigate this issue experimentally. We introduce several design decisions that altogether turn an existing hypervisor into a "passive" hypervisor, which interposes the operating system execution only to handle explicit security calls from it. We show that such a design can achieve performance close to that of native. ARM TrustZone used by the security monitor is already enabled by most current mobile devices and including this feature in the security monitor does not add any extra overhead. However, we also perform a study of the overhead of one of the TrustZone controllers by studying its hardware specification and demonstrate its overhead.

3.1 Virtualization's Performance

We start our investigation with an unmodified Xen hypervisor. We perform performance experiments in the operating system using the popular LMBench suite [143]. Our measurements show that Xen indeed incurs noticeable overhead to several microbenchmarks (up to 123.69%). We then perform detailed instrumentation to find out the sources of the performance overhead and remove them with several design decisions. In this section, we first introduce these design decisions. We then present the results of our measurements.

Design Decision I: No vCPU Scheduling. The hypervisor implements virtual CPUs (vCPU) on top of the physical CPUs (pCPU) in the system. It then assigns a configurable number of vCPUs to each virtual machine and schedules them. This scheduling incurs overhead. This is because the scheduler context-switches the vCPUs over pCPUs. On a context switch between two vCPUs, the scheduler stores the current vCPU's execution state (i.e., pCPU registers) in memory and restores the target vCPU's state. Moreover, to perform the context switch, the virtual machine execution traps into the hypervisor.

We observe that vCPU scheduling is only needed for multi-tenant virtualization environments, where the CPU resources need to be fairly shared between untrusting virtual machines. In a TEE design, transitions between the operating system and TEEs are mainly through explicit calls from the operating system. Therefore, no vCPU scheduling is needed. All the CPUs can be assigned to the main operating system. Each CPU can then switch to execute in a TEE if called by the operating system or to handle an interrupt from a secure I/O device (similar to transitions from the normal world to secure world with TrustZone).

We use two techniques to eliminate the vCPU scheduling. Our first technique is CPU pinning. That is, we pin the vCPUs to pCPUs. To do this, we allocate the same number of vCPUs for the operating system as the number of existing pCPUs in the system and we pin each vCPU to one pCPU. This prevents the hypervisor from performing any context switches.

Note that we will still be able to support explicit domain transitions. That is, if the operating system requires to invoke the TEE, it can issue a hypercall. The passive hypervisor can handle the hypercall by resuming the execution of the calling vCPU in the TEE. Once the request is handled, the vCPU will be programmed to continue its execution in the operating system. No vCPU scheduling will be performed.

Our second technique is to eliminate the use of the idle domain in Xen. When there is no running task, the operating system scheduler switches to the idle task. The idle task runs an idle loop, in which it calls the Wait-for-Interrupt (WFI) instruction. This instruction forces the processor to sleep and wait for an interrupt to wake up. With a commodity hypervisor, this instruction is configured to trap into the hypervisor, which then performs the idling by executing an "idle domain". This allows the hypervisor to make scheduling decisions, if needed. This trap, followed up by the execution of an idle domain, causes performance overhead. Therefore, we remove it in the passive hypervisor. That is, we deprivilege the WFI instruction, which prevents it from trapping. In this case, the operating system performs the idling itself, similar to an operating system running natively.

Design Decision II: Use Super Pages. One source of potential performance overhead is memory virtualization. In an ARM processor with virtualization hardware, there are two stages of address translations (guest virtual to guest physical and guest physical to system physical). A different set of page tables is used in each stage. The operating system controls the first set of page tables (i.e., stage-1 page tables) and the hypervisor controls the second one (i.e., stage-2 page tables). Therefore, the hypervisor can use different page sizes in stage-2 page tables. As we will show in §3.1.1, using the small page sizes of 4 KB causes noticeable performance overhead, for two reasons. First, for this page size, the page table walk consists of up to four lookups (hence four memory accesses). Second, such a page size results in more translation entries, which increases the Translation Lookaside Buffer (TLB) contention.

Therefore, our experiments show that it is critical to use super pages to implement the physical address space of the operating system. The question becomes: what super page size should the passive hypervisor use? ARMv8 processors support various super page sizes, e.g., 2MB, and 1GB. Requiring the use of 1 GB pages will minimize the performance overhead. However, it will prohibit us from launching more than a few domains, since the granularity of memory partitions is low. Therefore, we use 2 MB pages in the passive hypervisor (fortunately, this is Xen's choice as well). Our experiments show that this super page size results in close-to-native performance.

Design Decision III: No IPI Traps. A commodity hypervisor, such as Xen, interposes on Inter-Processor Interrupts (IPI), causing performance overhead. This interposition is needed in the hypervisor due to CPU virtualization. That is, when the operating system issues an IPI to another vCPU, it traps into the hypervisor, which then injects the IPI to the pCPU corresponding to the target vCPU. However, in a passive hypervisor, the vCPUs are statically mapped to pCPUs and hence this interposition can be eliminated.

We do not yet have this feature in our prototype, but here is how we plan to support it. ARM architecture has a hypervisor configuration register (HCR_EL2) that controls the traps to the hypervisor. HCR_EL2.IMO is the bit for routing physical interrupts to the guest or to the hypervisor. If this bit is unset, all the interrupts are routed to the guest removing the hypervisor from the interrupt path. While this is in principle possible and is our eventual prototype goal, it requires further rehauling of the hypervisor and possibly the guest, which is part of our ongoing efforts. However, in order to demonstrate the benefits of eliminating IPI traps in our benchmarks, we also measure the performance of our benchmarks on a single CPU (since such a configuration eliminates IPIs).

3.1.1 Virtualization Evaluation

We evaluate the effect of virtualization (both with commodity and passive hypervisors) on the main operating system. We use a HiKey development board, which has a Kirin 620 SoC with an octa-core ARM Cortex-A53 64-bit CPU operating at a maximum frequency of 1.2

Type	**Name**	**Native**	**Stdev(%)**	**Xen**	**Stdev(%)**	**Xen Ovr (%)**	**pXen**	**Stdev (%)**	**pXen Ovr (%)**
File Sys. BW. (ops/s)	mk	34657.58	0.91	34225.83	0.39	1.24	34298.08	0.07	**1.03**
	rm	51910.00	0.22	51755.25	0.18	0.29	51777.25	0.57	**0.25**
Syscall Lat. (us)	Simple	0.19	0	0.19	0	0	0.19	0	**0**
	rd.	0.60	0	0.60	0	0	0.60	0	**0**
	wr.	0.87	0	0.87	0	0	0.87	0	**0**
	stat	4.00	0	4.00	0	0	4.00	0	**0**
	fstat	0.85	0	0.85	0	0	0.85	0	**0**
	open/close	9.31	0	9.31	0	0	9.31	0	**0**
Select Lat. (us)	fd=250	14.68	0.13	14.76	0	0.54	14.71	0	**0.22**
Signal Lat. (us)	Installation	0.56	0	0.56	0	0	0.56	0	**0**
	Overhead	3.86	0	3.88	0	0.47	3.88	0	**0.47**
	Prot. fault	0.37	0	0.37	0	0	0.37	0	**0**
CPU Lat. (us)	Int64 bit	0.84	0	0.84	0	0	0.84	0	**0**
	Int64 add	0.08	0	0.08	0	0	0.08	0	**0**
	Int64 mul	3.34	0	3.35	0.17	0.29	3.35	0.17	**0.29**
	Int64 div	7.94	0	7.96	0.07	0.25	7.95	0	**0.12**
	Int64 mod	5.85	0	5.86	0	0	5.86	0	**0.17**
File read BW. (MB/s)	io_only	883.48	1.28	877.87	1.21	0.63	881.01	0.92	**0.27**
	Open2close	737.61	0.56	736.00	0.67	0.21	736.00	0.62	**0.21**
Mmap read BW. (MB/s)	io_only	2581.00	0.03	2539.66	0.58	1.60	2554.66	0.05	**1.02**
	Open2close	913.66	0.12	886.00	1.41	3.02	896.33	0.90	**1.89**
Memory Lat. (us)	load	107.73	0.03	109.83	0.63	1.95	109.70	0.72	**1.82**
Memory BW. (MB/s)	Read	2081.00	0.26	2056.00	0.12	1.20	2060.66	0.97	**1.00**
	Write	4665.66	0.01	4647.66	0.02	0.38	4654.33	0.02	**0.24**

Table 3.1: LMBench benchmarks that use one process. "Native", "Xen", and "pXen" columns show averages and the columns on their right show standard deviations. "Ovr" refers to overhead compared to native. Note that the results for memory benchmarks do include the effects of cache and TLB.

GHz [39]. Moreover, the board comes with 2 GB of memory. We build the passive hypervisor on top of Xen 4.9 hypervisor. We use CentOS with Linux kernel version 4.1 for the operating system. We did our measurements using the LMbench micro-benchmarks [143]. We evaluate three configurations: (*i*) native, in which there is no hypervisor and the operating system runs natively on top of hardware, (*ii*) Xen, in which we run the operating system on top of unmodified Xen, and (*iii*) pXen, in which we run the operating system on top of our passive hypervisor.

Tables 3.1 and 3.2 show the results. Table 3.1 shows the results for benchmarks that use a single process. Our results show that for these benchmarks, Xen does not incur noticeable overhead (an average of 0.50%). The passive hypervisor further reduces this overhead to an average of 0.37%. Table 3.2 shows the results for the benchmarks that use multiple processes. It shows that Xen does incur significant overhead for these benchmarks (due to vCPU scheduling and IPI traps). Moreover, it shows that in a multi-core environment

		Multi-core Experiments							
Type	**Name**	**Native**	**Stdev (%)**	**Xen**	**Stdev (%)**	**Xen Ovr (%)**	**pXen**	**Stdev (%)**	**pXen Ovr (%)**
IPC	Pipe Lat. (us)	15.22	4.79	31.85	1.63	109.16	15.34	1.16	0.76
	Pipe BW. (MB/s)	807.52	0.04	673.72	0.59	16.56	748.12	0.39	7.35
	Unix stream Lat. (us)	22.20	4.62	41.52	0.98	87.04	41.07	0.14	84.97
	Unix stream BW. (MB/s)	1988.14	1.74	1924.82	0.66	3.18	1962.99	0.32	1.26
Fork	Exit Lat. (us)	552.96	3.24	599.80	1.09	8.47	571.56	1.27	3.36
	Execv Lat. (us)	1538.33	1.76	1639.00	1.62	6.54	1634.16	0.24	6.22
	/bin/sh Lat. (us)	3303.33	2.50	3438.33	0.23	4.08	3341.16	0.35	1.14
Context switch Lat., s=4k, p=2-96 (us)		7.47	0.13	16.71	0.03	123.69	11.58	0.04	55.02
Netw. Lat. (us)	TCP	76.13	0.10	115.00	0.18	51.05	84.34	0.08	10.78
	UDP	59.27	0.32	76.77	1.51	29.51	67.71	0.25	14.22
		Single-core Experiments							
Type	**Name**	**Native**	**Stdev (%)**	**Xen**	**Stdev (%)**	**Xen Ovr (%)**	**pXen**	**Stdev (%)**	**pXen Ovr (%)**
IPC	Pipe Lat. (us)	23.11	0.46	23.20	0.46	0.37	23.14	0.42	**0.12**
	Pipe BW. (MB/s)	804.18	1.04	801.49	0.42	0.33	803.66	0.63	**0.06**
	Unix stream Lat. (us)	31.67	0.56	31.76	0.19	0.28	31.55	0.85	**-0.34**
	Unix stream BW. (MB/s)	915.03	0.39	909.32	1.53	0.62	918.00	1.19	**-0.32**
Fork	Exit Lat. (us)	494	0.20	501.33	2.37	1.48	501.15	2.34	**1.44**
	Execv Lat. (us)	1464.00	0.24	1492.00	0.52	1.91	1490.00	1.27	**1.77**
	/bin/sh Lat. (us)	3325.33	0.45	3434	0.48	3.26	3387.83	2.18	**1.87**
Context switch Lat., s=4k, p=2-96 (us)		11.43	0.10	11.64	1.72	1.80	11.56	1.40	**1.16**
Netw. Lat. (us)	TCP	87.61	0.24	87.92	0.13	0.35	87.88	0.16	**0.31**
	UDP	61.52	0.14	61.87	0.27	0.57	61.67	0.13	**0.25**

Table 3.2: LMBench benchmarks that use multiple processes. "Native", "Xen", and "pXen" columns show averages and the columns on their right show standard deviations. "Ovr" refers to overhead compared to native.

(Table 3.2, top), the passive hypervisor (pXen) achieves noticeably better performance compared to Xen due to its elimination of vCPU scheduling. For example, in the context switch benchmark, Xen has an overhead of 123% while the passive hypervisor has an overhead of 55%. The remaining overhead is because of IPIs, which happen in a multi-core environment. As mentioned in §3.1, we currently do not support the elimination of IPI traps. Therefore, to demonstrate the effect of this elimination, we also run our benchmarks in a single-core environment (Table 3.2, bottom) in order to avoid IPI traps. In these tests, we pinned the benchmarks' processes to a single CPU in the system. Our results show that without the IPI traps, the passive hypervisor overhead can be reduced significantly (to an average of 0.63% across benchmarks).

Memory latency: We also measured the effect of the page size used in the stage-2 page tables on the memory latency. Our experiments show that the average memory load latency is 109.83 us and 158.13 us with the 2 MB and 4 kB page sizes, respectively. This shows that it is critical to use super pages in the stage-2 page tables, as mentioned in §3.1.

3.2 TrustZone's Performance

One of the components in SoCs with TrustZone support is TrustZone Address Space Controller (TZASC). TZASC performs security checks on every read or write accesses to memory or I/O devices. These checks are performed using filters. Each filter controls memory accesses by a single or multiple sources (i.e., CPU and DMA engines) and it can be set up to control up to 8 separate regions. Each region is a contiguous range of memory addresses. For every transaction, the controller looks at all the regions in the filter. If the transaction address matches the address range of the region, the controller checks whether the access is allowed based on the access type (secure/non-secure read/write access) and the access permissions set on the region.

We study TZC-400, a popular TZASC. Our study shows that the controller has a minimum 2 cycles overhead for each memory write access. This additional overhead impacts both memory accesses and I/O devices (register accesses as well as Direct Memory Access (DMA)). For example, for memory writes, assuming a 107 ns memory access latency (Table 3.1) and a max frequency of 1.2 GHz (as in our prototype described in §3.1.1), the TrustZone overhead is about 1.5%. However, note that with memory virtualization, the overhead is due to address translation, which can be reduced with a good TLB algorithm. However, with TrustZone, the overhead is unavoidable since the security checks are performed on the physical addresses.

Note that memory reads are not mostly affected since the controller supports speculation access [71] (i.e., the transaction is dispatched to memory in parallel to the security checking).

However, TZC-400 supports only 32 in-flight speculative transactions; thus, a memory-heavy benchmark might experience overhead for memory reads as well.

Chapter 4

Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems

Modern mobile systems such as smartphones, tablets, and wearables have one important property in common: they contain a plethora of sensors. A smartphone today, for example, contains tens of sensors including camera, microphone, GPS, accelerometer, and fingerprint scanner. Moreover, being mobile, these systems are with the user at all times. Such a usage model exposes mobile sensors to extremely private information about the user, including daily conversations, photos, videos, and visited locations, all of which are considered to be sensitive by some or most users according to a study by Muslukhov et al. [153].

Such a powerful sensing capability raises important privacy concerns. These concerns are reinforced given the incidents where women were spied on through their webcams [12, 14], and users were spied on through their smartphone microphones and cameras [10]. In fact, in Android, the camera and microphone were shown to be hacked and controlled remotely using a commercialized trojan [6].

We believe that an important and practical remedy to this problem is to provide *trustworthy sensor notifications* on mobile systems in order to provide *unconditional and immediate feedback* to the user when the sensors are being used. That is, we believe that mobile systems must use an *indicator*, such as LED or vibrator, to unexceptionally notify the user when a sensor is on. This way, even if the sensor is accessed maliciously, the user becomes aware and can take action, e.g., by turning the system off. The usefulness of trustworthy sensor notifications is due to the fundamental observation that the user is able to reason about the correct status of a sensor at a given time. For example, the user correctly expects the camera to be on only when she explicitly launches a camera application. Similarly, she expects the microphone to be recording only if she launches a voice recorder application or if she makes a voice call. With such knowledge, the user can leverage trustworthy sensor notifications to detect malicious access to sensors.

Unfortunately, sensor notifications are not systematically enforced in mobile systems today and existing notifications are ad hoc. First, some applications implement their own notifications. For example, the built-in microphone application on Samsung Galaxy Note 3 blinks a blue LED while recording if the display is turned off. It also adds an unremovable glyph to Android notification bar when the application is in the background. Or the built-in camera application in the same smartphone only records video if it is in the foreground with the display on (which can be considered as some form of notification). However, these notifications do not apply to other applications. For example, the Detective Video Recorder application [1] can record video and audio when in the background and with the display off. Second, laptop webcams often notify the user with a built-in LED. These notifications are implemented by the webcams themselves and do not apply to other sensors. Moreover, as Brocker et al. [82] showed, these notifications can be circumvented in some MacBook

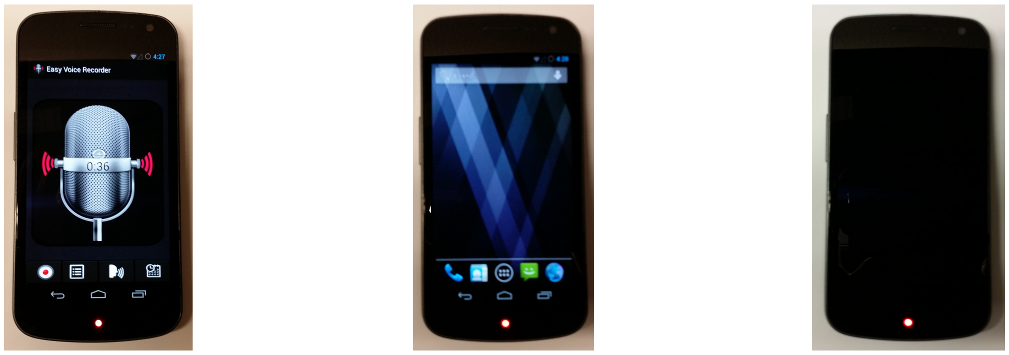


Figure 4.1: LED notification for microphone using Viola. The LED blinks even if the application is in the background or if the display is off. Malicious attempts to break this notification will be blocked.

laptops and iMac desktops by rewriting the webcam firmware, demonstrating the difficulty of implementing trustworthy sensor notifications.

In this chapter, we present *Viola*, a system solution for providing trustworthy sensor notifications in mobile systems. Our fundamental observation is that it is possible to formulate a sensor notification as a logical *invariant* on the hardware states of the sensor and indicator. For example, for an LED notification for microphone, the invariant is `microphone recording` → `LED blinks`, where → depicts logical implication. To do so, Viola employs runtime invariant checks as a security service in the security monitor to guarantee that this invariant is never violated even if the operating system is compromised by an attacker. As we discussed in chapter §2, inserting the checks in the low-level system software enhances the trustworthiness of Viola as it reduces the size of the TCB. In this design, bugs in the device drivers and I/O services, which are very common [94, 106, 156, 8, 80], will not undermine the sensor notification invariants. Moreover, this design protects the integrity of the notification invariant against powerful malware including those with root or kernel privileges. That is, the check does not allow the microphone to start recording unless the LED is blinking and does not allow the LED to be turned off while the microphone is recording. Figure 4.1 shows Viola in action. See a video demo in [22].

In this chapter, we also answer an important question about the design of the invariant checks.

Q. How can system designers develop provably correct invariant checks? Writing error-free invariant checks that operate on the parameters of writes to registers is fairly complicated given the complex hardware interface of many I/O devices, their peripheral buses, and the components they rely on, such as power supplies and clock sources in a System-on-a-Chip (SoC).

We tackle this challenge using an invariant language and its verified compiler. Viola's invariant language enables the developer to mainly focus on the invariant logic using a high-level and intuitive syntax. A verified compiler, which we build and verify using the Coq language and its proof assistant [20], guarantees that the generated invariant checks preserve the semantics of Viola's invariant language. The compiler uses device specifications for inferring device state transitions as a result of register writes.

We present an implementation of Viola that supports different sensors and indicators, such as camera, microphone, LED, and vibrator, on two smartphones, Nexus 5 and Galaxy Nexus. While hardware support for virtualization is increasingly available on mobile systems [70], commercial mobile systems either do not leverage this hardware support or, if they do, they do not provide open access support for programming the hypervisor. Therefore, to demonstrate the feasibility of using the hypervisor layer, we implement Viola on the ODROID XU4 development board as well. We have open sourced the implementation of Viola [21].

We demonstrate that implementing sensor notifications using Viola does not require significant engineering effort. We also demonstrate that Viola adds significant latency to every monitored registered write (especially if Viola's monitor runs in the hypervisor) but that this latency incurs almost no overhead to the sensor's performance due to the infrequency of monitored register writes. Moreover, we show the additional power consumption is small.

Note that Viola does not address all the privacy concerns that a user may have with respect to the sensitive information captured by mobile sensors, including unauthorized access to already-captured information. Moreover, Viola does not tell the user which application is using a sensor. It just informs the user that the sensor is being used relying on the user to decide whether the access is malicious or not and, if yes, to detect the malicious application. Addressing these concerns is orthogonal to our work.

Also note that Viola requires modifications to various parts of the system software. This is because Viola is implemented in low-level system software to build a secure system that is protected from the operating system and applications. Hence, it is not easily deployable by ordinary users on their mobile systems, e.g., by installing an application. We mainly envision Viola to be adopted by mobile system vendors, such as Samsung and LG. However, we believe that expert users can also benefit from Viola especially since Viola's verified compiler makes it easy for them to develop the required invariant checks. Note that an easily-deployable solution is inherently vulnerable to tampering by malware. This is due to a fundamental trade-off between how easily the system can be deployed and how secure the system is against malware.

4.1 Sensor Notifications

4.1.1 Indicators

Various indicators including LED, vibrator, speaker, and display can be used on mobile systems today. Here, we discuss the pros and cons of these indicators.

LED. Today's mobile systems incorporate several LEDs on various locations on the system's exterior, e.g., on top or bottom of the display. LEDs come in different colors and can be used in different modes, e.g., constant illumination and blinking. LEDs provide the most lightweight indicators possible and we anticipate them to be the most dominant indicators for many sensors. However, LEDs have two important shortcomings. First, they are ineffective if the system is out of user's sight, e.g., in the user's purse or pocket. Second, they are often overloaded with informing the user of other events in the system as well (e.g., a missed call or text message), which reduces their effectiveness in attracting the user's attention.

Vibrator and speaker. These two indicators are effective in capturing the user's attention even when the system is out of the user's sight. However, they can be intrusive and distracting especially if used for extended periods of time. Moreover, they pollute the data captured by some sensors, e.g., microphone and accelerometer.

Display. Display is another channel to convey notifications to the user. However, not only this notification requires the display to be in user's sight, it also requires the display to be on, which is power hungry.

Note that the best notification might be achieved by leveraging several indicators. For example, for microphone, it might be best to play a short beeping sound on the speaker and illuminate an LED afterwards.

Also, note that our goal in this paper is to build a trustworthy framework for notifications. In an orthogonal work, we have performed a user study to understand the characteristics of various indicators in mobile systems [137]. In this user study, we show how effectively each indicator can grab user's attention in different contexts. A similar study exists for laptops [158].

4.1.2 Notification Guarantees

Causal guarantees. We envision two types of notifications in terms of causal guarantees provided to the user. *One-way* notifications guarantee that the indicator is on, e.g., the LED is illuminated, when the sensor is on. However, they do not provide any guarantees on the state of the indicator when the sensor is off. The logical invariant for one-way notifications can be expressed as `sensor in target state` $\rightarrow$ `indicator in target state`. For a simple uni-color LED, the target state can be as simple as LED being illuminated. For an RGB LED with built-in blinking capabilities and varying illumination levels [18], the target state may be that the LED blinks in a red color with a given frequency and illumination level. In the rest of the paper, we use the terms "in target state" and "on" interchangeably.

Two-way notifications provide both guarantees. That is, they guarantee that the indicator is on *if and only if* the sensor is on, or `sensor on` $\leftrightarrow$ `indicator on`. With one-way notifications, the indicator can be turned on while the sensor is off, resulting in annoyance and false notifications to the user. By triggering a large number of false notifications, an attacker can reduce the effectiveness of the notification as the user will likely ignore the notifications due to fatigue. Two-way notifications eliminate the false notifications and hence solve this problem.

Note that Viola uses a specific target state (i.e., operation pattern) of an indicator for a sensor, e.g., an LED blinking with a certain color and frequency. Therefore, other applications and system services can still use the indicator, but with different operation patterns.

Temporal guarantees. We envision two types of notifications in terms of the duration in which the indicator is active. *Continuous* notifications guarantee that the indicator to be on for as long as the sensor is recording and are best suited for LED and display indicators. *Temporary* notifications guarantee the indicator is turned on only for a fixed period of time when the sensor starts capturing and are best suited for speaker and vibrator indicators.

In this paper, we demonstrate the use of LED and vibrator as indicators while supporting one-way, two-way, and continuous notifications. In [147], we have discussed the challenges of providing formal guarantees for the use of speaker and display as indicators and for temporary notifications.

4.1.3 Customizable Sensor Notification

It is critical that the sensor notifications cannot be deactivated by attackers. This has motivated us to design a solution that enforces the notifications unconditionally. However, we envision scenarios that the user might need to disable the notifications temporarily, e.g., if light, vibration, or sound interferes with user's activities. Allowing customizations while protecting against malware raises new challenges. In[147], we have discussed some ideas for addressing these challenges.

4.2 Overview

Figure 5.1 illustrates the design of Viola. There are three main components. The first component is the implementation of sensor notification, which leverages the device driver (either directly or indirectly through user space I/O service API) to turn on the indicator before the sensor starts recording and to turn the indicator off after the sensor stops. This implementation is not trusted and hence errors in it will not break the notification invariant. Malfunction in this implementation, e.g., failure to turn on the indicator, will be detected and blocked by the invariant checks (the third component, explained below).

The second component is the security monitor. In Viola, we also design a kernel-based security monitor that provides the functionalities of the security monitor in the kernel rather than the hypervisor. It intercepts the device drivers' attempts to write to the registers

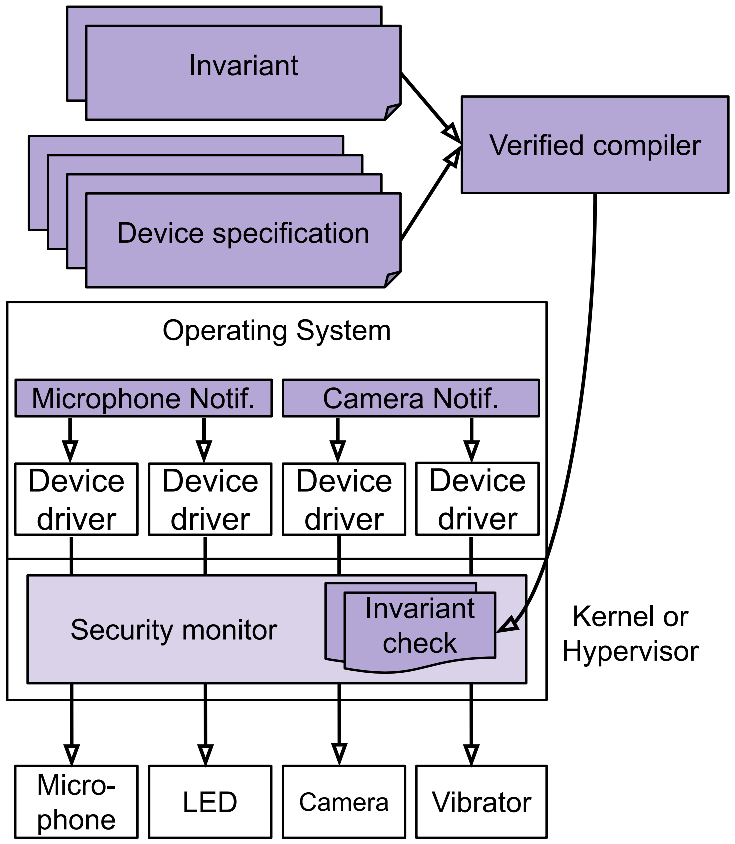


Figure 4.2: Viola's design. The darker components belong to Viola.

of sensors and indicators by removing the write permissions from the corresponding page table entries. It then consults with a set of deployed invariant checks, which inspect the parameters of the write and decide whether, if allowed, the write would break the invariants corresponding to the sensor notifications or not. The monitor allows for the successful execution of the write only if it passes all the checks, and rejects it otherwise. In case of a reject, it blocks the write and force-reboots the system. §5.3 provides more details on the monitor.

The third component of our design is a set of invariant checks deployed in the security monitor. To enable developing provably correct invariant checks, we present a high-level language for writing the sensor notification invariants. We also develop a verified compiler for the language using Coq that guarantees that the generated invariant checks maintain the semantic of sensor notification invariant. The compiler uses device specifications for inferring the device state transitions as a result of register writes. In addition, for off-chip devices whose registers are only accessible through a peripheral bus, we design a verified bus interpreter module that can infer the device register writes from the bus adapter register writes. §4.4 discusses the details of the invariant language and its compiler.

4.2.1 Threat Model

We have designed Viola to enforce sensor notifications' invariants in the system and prevent buggy or malicious code from breaking them. We assume attackers with varying capabilities. The first attacker can only use the application API in the operating system, e.g., Android API. This attacker can run native code as well but without root privilege. The second attacker runs native code with root privilege in the user space, however cannot run code with kernel privilege or secretly modify the system image (for future boots). More specifically, in this case, we assume that the attacker cannot leverage the kernel vulnerabilities to inject code and assume that the kernel is configured to prevent a root user from easily modifying the running kernel memory. The latter is achieved by configuring the kernel to disallow loading of kernel modules and to avoid exposing the `/dev/kmem` file, which would allow the user space code to have access to the kernel memory. Moreover, we assume that the mobile system implements the verified boot feature [4], which checks and verifies the integrity of the loaded system images. As a result, the attacker's attempt to modify the system images (i.e., the kernel and hypervisor images), which will take effect in future boots, will be detected and then blocked or at least communicated to the user with a notification at boot time. The third attacker leverages the vulnerabilities of the kernel to compromise it and hence can run code with kernel privileges. The fourth attacker is a more advanced version of the third attacker that leverages the vulnerabilities of the hypervisor to compromise it and hence can run code with hypervisor privileges. The fifth attacker is a root user in a system without the verified boot feature, which would allow him to rewrite the kernel and hypervisor images (to be used after a reboot). Finally, the sixth attacker has physical access to the device and can manipulate the hardware.

As mentioned earlier, we run Viola's monitor either in the kernel or the hypervisor. In both cases, Viola enforces the integrity of the sensor notifications despite bugs in the I/O stack (including device drivers and user space I/O services), which would otherwise violate the notifications' invariants. Moreover, both types of monitor protect against the first two attackers. However, only the hypervisor-based monitor protects against the third attacker. This attacker can circumvent the kernel-based monitor in three ways: (*i*) it can manipulate the fault handler in the kernel in order to prevent the deployed invariant checks from evaluating a register write, (*ii*) it can access the registers through a different set of virtual addresses mapped to the same register pages, or (*iii*) it can manipulate the page tables to re-enable the write permissions on the page table entries removed by Viola's monitor. None of the solutions can protect against the fourth attacker. This attacker is a super set of the third attacker and hence can circumvent the kernel-based monitor as explained above. Moreover, it can use very similar techniques to circumvent the checks implemented in the hypervisor. Similarly, none of the solutions can protect against the fifth attacker since it can simply remove Viola's monitor from the system image used for future system boots. Finally, none of the solutions can protect against the sixth attacker as it can modify the hardware, e.g., disconnect the LEDs.

4.3 Runtime Monitor in Viola

Runtime monitor runs in the low-level system software, e.g., in the operating system kernel or the hypervisor, in order to monitor the device drivers' interactions with sensors and indicators. The monitor intercepts any attempts to write to device registers and feeds the write parameters to the invariant checks (§4.4). It allows the successful execution of a write only if it passes all the checks. Note that we focus on register writes, and not register reads, since the former is typically used to alter the state of a device. However, all the discussions are easily extended to register reads if they need to be monitored as well.

Using the kernel or hypervisor to insert the invariant checks provides a trade-off between trustworthiness, support for different operating systems, support for legacy systems, and performance. On the one hand, a hypervisor-based solution isolates the monitor from the operating system, which enhances its trustworthiness even against malware with complete control over the kernel execution (i.e., the third attacker in §6.7.1). It also makes the monitor implementation agnostic to the operating system, e.g., Android or iOS, and its version. The use of the hypervisor is further motivated by the fact that ARM processors have recently added virtualization hardware support [70] allowing an efficient implementation of the monitor in the hypervisor.

On the other hand, a kernel-based solution can be used in many existing mobile systems. This is because commercial mobile systems either currently do not have hardware support for virtualization or do not provide open access for programming the hypervisor. Also, inserting the checks in the kernel incurs less overhead to a register write compared to a hypervisor-based solution (§4.7.2).

Viola's monitor intercepts device drivers' attempts to write to registers by removing the write permissions from the corresponding page table entries. This is feasible since in the ARM architecture, all the registers are memory-mapped (i.e., Memory-Mapped I/O or MMIO). Depending on where the monitor is hosted, we use different page tables. First, for the hypervisor-based monitor, we leverage ARM's Stage-2 page tables [97, 70]. In this case, a memory address is translated by two sets of page tables, one maintained by the operating system and one (i.e., Stage-2 page tables) maintained by the hypervisor. Removing the write permission from a Stage-2 page table entry forces writes to every register in the corresponding page to trap in the hypervisor, allowing the monitor to inspect them. Second, for the kernel-based monitor, we simply use the page tables used by the kernel. In this case, a write to protected register pages raises a page fault exception in the kernel.

When the monitor detects an unauthorized register write, it blocks the write and force-reboots the system. It does so by returning a permission violation error to the page fault handler in the kernel or hypervisor, which leads to a reboot.

Initially, we planned to identify and kill the responsible application upon detecting such page faults. However, doing so is challenging, as the violation might be triggered not by an application but by powerful malware, e.g., one with kernel privilege. Therefore, we believe that force-reboot is a safe approach to handle these attacks. Note that the force-reboot approach does not create an easily-exploitable and additional denial-of-service attack vector. This is because attackers with root, kernel, or hypervisor privileges can reboot the system using other methods anyway and the attacker without these privileges (i.e., the first attacker in §6.7.1) cannot trigger such page faults easily unless it circumvents the implementation of the sensor notification.

4.3.1 Protection Against Concurrency Attacks

Sensors and indicators can be accessed concurrently by multiple threads. Therefore, our runtime monitor can be executed concurrently by these threads. The execution of the runtime monitor consists of checking the sensor notification invariant and changing the state of sensor or indicator (Figure 4.3 (Left)). In doing so, the monitor maintains a global data structure that captures the current state of the sensor and indicator (i.e., the value of each register) and updates it when emulating a register access. This data structure is then used to determine the legality of the subsequent register accesses. Therefore, to ensure the integrity of this data structure, we execute the runtime monitor in a critical section. This is because concurrent execution of the monitor may corrupt this global data structure. For example, consider the

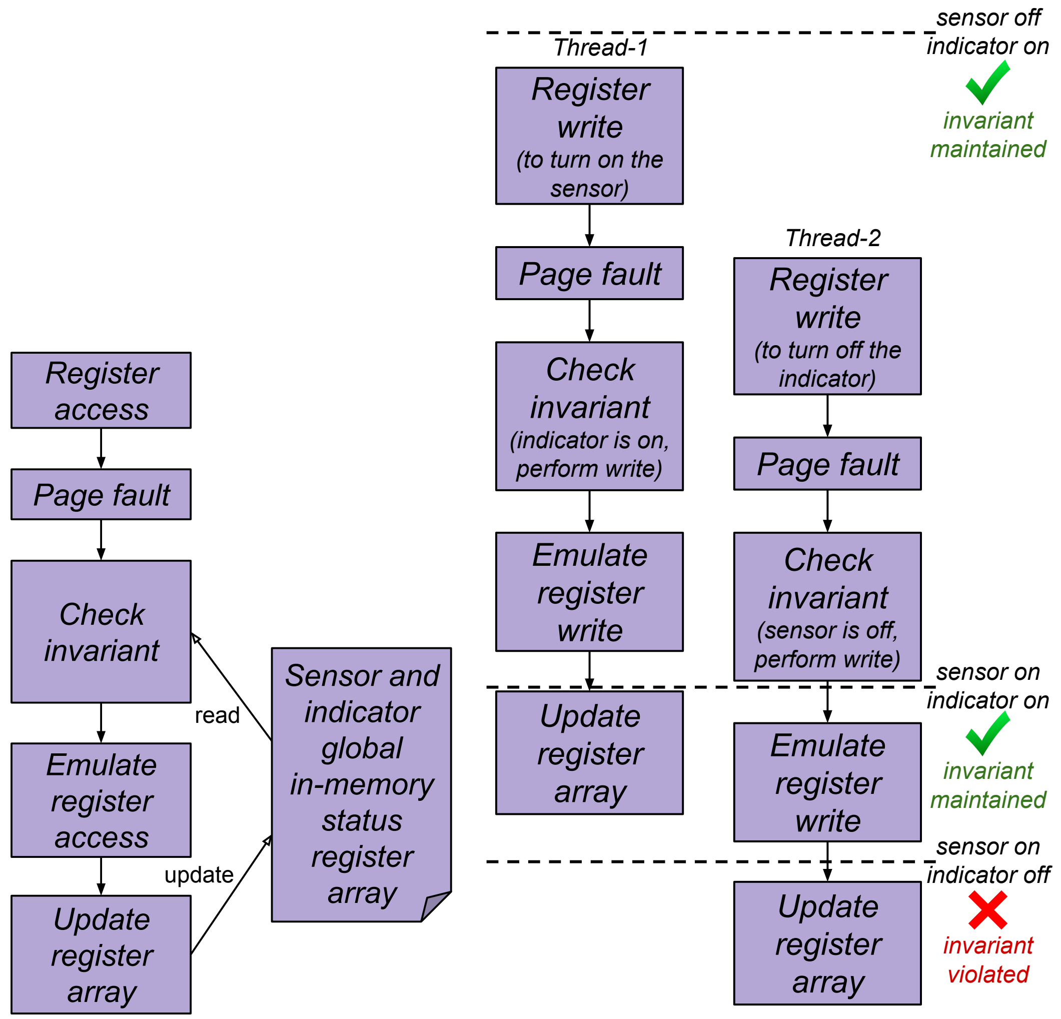


Figure 4.3: (Left) The sequence of steps in handling a register write in Viola. (Right) A concurrency attack example in Viola.

`sensor on` $\rightarrow$ `indicator on` invariant and assume that initially the sensor is off and the indicator is on. The following interleaving of accesses by two different threads will end up in a violation of the notification invariant (as also demonstrated in Figure 4.3 (Right)):

1) Thread-1 attempts to turn on the sensor with a register write access. The runtime monitor gets triggered; it checks the invariant and approves the access since the indicator is on.

2) About the same time, Thread-2 attempts to turn off the indicator with a register write access. The runtime monitor gets triggered; it checks the invariant and approves the access since it sees that the sensor is off (note that the runtime monitor checking the access by

Thread-1 has not yet updated the global data structure). It then emulates the register write access, which turns off the indicator, and then updates the global data structure reflecting this.

3) The runtime monitor handling Thread-1's register access continues its execution simultaneously; it emulates the previously approved sensor register write access, which turns on the sensor, and then updates the global data structure reflecting this.

As can be seen, this interleaving of events results in the sensor turning on while the indicator turns off, which violates the invariant that the monitor is supposed to enforce. Our previous report of Viola [147] tried to address this problem by piggy-backing on the critical section enforced by the fault handler, e.g., the kernel fault handler. Page faults triggered by threads in the same address space (i.e., process) execute in a mutually exclusion manner. However, an attacker can try to circumvent this critical section by using two threads belonging to different processes. To further protect against such concurrent attacks, Viola uses a global mutex lock to serialize the monitor's handling of register accesses, no matter which thread attempts them. One might be concerned that adding a mutex lock to the monitor's handling of all register accesses might have significant overhead on Viola's performance or on the system power consumption. In §4.7.2 and §4.7.3, we show that performance and power overheads are negligible and small, respectively.

4.4 Verified Invariant Checks

Viola's verified invariant checks receive the parameters of a register write as input from the monitor and decide whether the write, if executed, breaks any of the sensor notifications' invariants in the system. Unfortunately, manually developing checks that operate on the parameters of register writes is cumbersome and error-prone. An I/O device, e.g., a sensor or an indicator, might have several registers, each affecting the behavior of the device in some way. Moreover, a single register might contain a few device variables (i.e., the variable formed by a subset of the bits in a register [144]), requiring bitwise operations in the invariant checks. Furthermore, the correct behavior of an I/O device might depend on other components as well, such as its power supply and clock source in the SoC, requiring Viola to monitor these components as well. Finally, Viola's monitor cannot directly monitor the writes to device registers for off-chip devices. It can only monitor the writes to registers of the peripheral bus adapters and must infer the device register writes.

In this section, we present our solution that enables the system designer to easily develop provably correct invariant checks. Our solution is an invariant language with a high-level and intuitive syntax, which enables developers to mainly focus on the invariant logic (which is quite simple), rather than the low-level implementation. We then present a verified compiler for this invariant language that generates provably correct assembly code that maintains the logical semantics of the invariant.

Viola considers the hardware dependencies, such as power supply and clock source, by using additional verified checks. It also uses a *bus interpreter module*, compiled from high-level "rules" using another verified compiler, to infer the device register access parameters from those of the bus adapter. We refer the reader to [147] for more details on these two issues.

4.4.1 Viola's Invariant Language Syntax

Viola's invariant language leverages two main programming constructs. (*i*) *State* is used to specify the target states of the sensor or indicator, e.g., microphone recording or LED blinking. A device target state is a list of device variable states, where each device variable state is a 2-tuple containing a device variable and a value. As mentioned earlier, a device variable is the variable formed by a subset of the bits in a register [144]. As a hypothetical

```
Definition mic_rec :=
    State mic_spec [mic_bias_on;   ...].
Definition led_blink :=
    State led_spec  [led_blink_mode;
                          led_color_red ;   ...].
Definition mic_led_notif :=
    Binder mic_rec led_blink .
```

Figure 4.4: The implementation of a microphone LED notification in Viola. For brevity, we have not shown the complete array of device variable states and the device specifications used in the code.

example, the three least significant bits of a register could form the device variable for camera resolution. These device variables must be defined in the device specifications (§4.4.2). (*ii*) *Binder* is used to bind the target states of the sensor and indicator.

Figure 5.5 shows an example invariant written in Viola for the microphone LED notification (LED blinking). In this example, we have first defined the target states of the sensor and indicator, `mic_rec` and `led_blink`, respectively. Each of these target states is defined as an array of device variable states, e.g., `led_blink_mode`. Moreover, in defining these target states, we have used the device specifications, i.e., `mic_spec` and `led_spec`, which provide the definition of each of the device variable states. Finally, the sensor notification invariant is defined by binding the target states of the sensor and indicator together.

4.4.2 Verified Compiler

We build a verified compiler for Viola's invariant language. Given an invariant written in this language, the compiler generates provably correct invariant checks to be inserted in Viola's monitor.

Building and verifying a compiler is a cumbersome task. Our key idea to reduce this effort is to implement our compiler as a frontend for the formally verified CompCert C compiler [126], similar to the approach used by Wang et al. for Jitk [174]. The frontend compiles Viola's code to Cminor, an intermediate language in CompCert, which is then compiled to assembly by

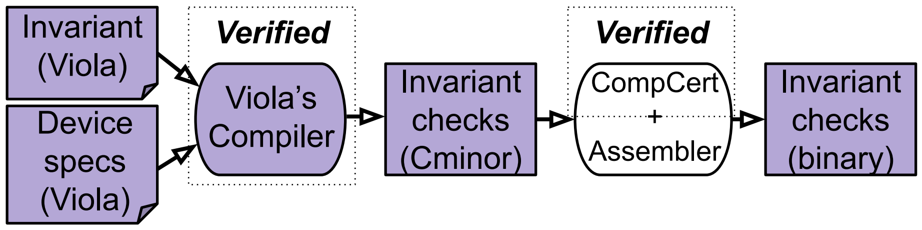


Figure 4.5: Compilation of Viola invariant code. Viola's compiler translates Viola code to Cminor code, which is then compiled to machine code using the verified CompCert compiler and an assembler.

the CompCert backend. The assembly code is then translated to machine code using a commodity assembler since CompCert does not currently provide a verified assembler. Cminor is a simplified and typeless variant of a subset of C with support for integers, floats, pointers (but not the & operator), control constructs such as if/else (but not the goto statement), and functions (see [126] for more details).

Given that CompCert is a verified compiler, it preserves the semantics of the code in Cminor while generating the assembly code. Therefore, we only need to prove that our frontend maintains the semantics of Viola's code while generating the Cminor code. To achieve this, we implement the frontend in Coq and use Coq's interactive proof assistant [20] to prove the semantic preservation property. As will be explained, our compiler relies on device specifications to infer the device state transitions as a result of register writes. Figure 4.5 illustrates our approach.

We prove the semantic preservation property as follows. We formalize the I/O devices (i.e., sensors and indicators) as blocks of memory (MMIO) corresponding to their register spaces. We then prove, through forward simulation [135, 136], that both programs (i.e., the one written in Viola and its compiled Cminor program) return the same decision (i.e., allowing or rejecting a register write) given the state of these MMIO blocks, the invariant logic, and the device specifications. Put formally, we prove the following theorem.

Theorem *Viola_compile_correct:* `forward_simulation (Viola.semantics Vprog)`

`(Cminor.semantics Cprog).`

The heart and the majority of the proof for the aforementioned theorem is proving the following lemma (slightly simplified for clarify).

Lemma *compile_step:* ∀ `S1 S2, Viola.step S1 S2` → ∀ `R1, match_states S1 R1` →
∃ `R2,`
`plus Cminor.step R1 R2` ∧ `match_states S2 R2.`

Both languages, Viola and Cminor, are modeled as a series of steps from some initial state to some final state. This lemma mentions that, for every two states in Viola code, for which there is a transition (i.e., a step) according to the semantics of the language, and for all Cminor states equivalent to the source state in Viola code, there exists a state in Cminor code that can be reached from this equivalent source state, possibly in multiple steps (as signified by `plus`), and that the reached target state in Cminor code is also equivalent to the target state in Viola code. The states and steps in Viola are part of the specification of the language semantics. Moreover, the definition of equivalence between the states of the two programs is part of the specification that we develop for the proof.

In Viola, each step is equivalent to the evaluation of one invariant check on the register write parameters. There are two main types of steps: reject and pass. In the former, the invariant check fails and Viola returns reject. In the latter, the invariant check passes and Viola continues to evaluate the next check. If all checks pass, Viola returns pass.

Reject or pass steps are determined based on *reject and pass conditions* that we have also defined as part of the language semantics. The reject condition is satisfied when either of its two sub-conditions are satisfied. The first sub-condition is when the register write causes the sensor to transition to its target state and the indicator is not in its target state. The second sub-condition is when the register write causes the indicator to go out of its target state and the sensor is in its target state. We specify transitioning to the target state as when all the device variables acquire the values defined in the device target state. We specify transitioning out of the target state as when at least one device variable acquires a value other than the one defined in the device target state. The pass condition is simply the negation of the reject condition.

In addition to this lemma, we also prove that for all initial states in Viola program, there is an equivalent initial state in the compiled Cminor program and that if the two programs reach equivalent final states, they both return the same value.

In proving the aforementioned theorem, we prove the soundness and completeness of the generated checks: that is, we prove that the generated checks always reject the register writes that violate the invariants (soundness) and never reject benign register writes (completeness).

Device Specifications

The compiler uses the device specifications to generate the invariant checks. The compiler needs these specifications for *device state transition inference* upon a register write. The specification must contain the following definitions: the list of registers (in the form of the register address offset from a base address), the list of device variables built on top of each registers, the list of device variable states, the list of register write events for the device variables, where an event is a write with a given value to a device variable, and the list of device variable transitions, where a transition is a 3-tuple consisting of a source state, an event, and a target state. Note that checking the source state of a transition in the invariant check is important in order to enforce the order of register writes. As a simple example, consider a device with a single 8 bit register, which consists of only one single-bit device variable. The list of register write events can be writes with the values of 0 and 1. The list of device variable transitions are transitions from 0 to 1 (with a write with the value of 1) and from 1 to 0 (with a write with the value of 0). For brevity of writing the events, Viola supports two types of events representing a write to a device variable with a value equal to or different from a specified value. This significantly reduces the size of the specification as it alleviates the need to enumerate all the events one by one. Note that while Viola enforces the order of register writes, it currently does not support specifying and enforcing the maximum time interval between consecutive register writes.

Currently, we manually develop these specifications for the devices that we support. Fortunately, as observed by Ryzhyk et al. [161], device specifications are increasingly available from hardware vendors as the same specifications are also used in the device design process. However, to leverage these specifications, a translator is needed to transform the existing specifications into the format supported by Viola.

Partial device specifications. Writing the complete device specifications can be a daunting task, especially for complex devices such as camera. However, in Viola, partial specifications suffice for two reasons. First, one does not need to develop the complete specification of the sensor; the specification of a few device variables used for turning on the sensor is adequate. As long as the invariant check guarantees that these device variables will take on their target value only when the indicator is on, the sensor notification invariant will not be violated. Second, while a more elaborate specification is often needed for the indicator, still a complete specification might not be needed. For example, if an LED is required to illuminate constantly, and not blink, one does not need to write the specifications for the blinking functionality of the LED.

4.5 Two-way Notification

As we discussed in §4.1, a two-way notification (`sensor on` $\leftrightarrow$ `indicator on`) provides a guarantee that the indicator is on if and only if the sensor is on. That is, it guarantees that the states of both the sensor and indicator change simultaneously. For example, for a two-way notification of microphone as sensor and LED as indicator, LED cannot be turned on by an attacker if the microphone is off and the microphone cannot be turned on if the LED is off. Compared to one-way notifications, two-way notifications provide a key advantage. That is, two-way notifications avoid false notifications to the user since the notification is only triggered when the sensor is actually used.

Eliminating false notifications is critical for two important reasons. First, it avoids annoying the user of the mobile device with an excessive number of notifications. In a user study that we recently performed [137], we found out that some notifications create annoyance for the users. For example, vibration and sound are generally described as annoying by users especially for certain sensors such as camera. With a one-way notification, an attacker can trigger the indicator for a long period of time causing annoyance for the user, potentially forcing him/her to disable the notification (if possible). With a two-way notification, these extra notifications are blocked and the user will not get distracted and annoyed.

Second, excessive notifications might reduce their effectiveness. That is, the user might stop paying attention to the notifications if they are triggered very often with a large number of false positives. Therefore, the attacker can try to make the user insensitive to a given sensor notification, by triggering a large number of false notifications, before he accesses a sensor.

4.5.1 Straw-man Approaches

We first explain two straw-man approaches to implement two-way notifications and highlight their shortcomings.

The first solution attempts to accept or reject the register writes in a way that guarantees that the two-way invariant is not violated. This approach is similar to how one-way notification was achieved. That is, the monitor intercepts all the register writes of interest on the sensor and the indicator and rejects those that violate the invariant. Unfortunately, this approach cannot enforce the two-way invariant. This is because making "local" decisions on single register writes will result in the violation of the two-way notification unless both the sensor and indicator are always off.

For example, consider a scenario where an application tries to turn on the sensor. In order to enforce the two-way notification, Viola's notification implementation can try to turn on the indicator first and then proceed to turn on the sensor. The Viola's monitor will then intercept these two writes, the first one attempting to turn on the indicator and the second one trying to turn on the sensor. It is easy to show that there is no feasible decision pair (one for each write) that satisfies the two-way notification other than two rejects (which would keep both the sensor and indicator off making them unusable). If the monitor accepts the first write and rejects the second one, the indicator is turned on while the sensor is still off, hence violating the invariant. If it rejects the first one and accept the second one, the sensor is on while the indicator is off, again violating the invariant. Finally, if it accepts both, in the time gap between turning on the sensor and the indicator (which can end up being long depending on the operating system thread scheduling), the two-way notification guarantee is violated.

The second straw-man solution is to make local decisions on every register write, but make sure that the sensor and indicator are turned on/off within a fixed period of time (e.g., 1 ms). Unfortunately, providing such a guarantee in practice is challenging because (*i*) the monitor cannot guarantee (on its own) that the two devices are eventually turned on/off within that period of time, and (*ii*) one cannot easily determine an acceptable period of time (between turning on/off the two devices) that works for all sensors and indicators.

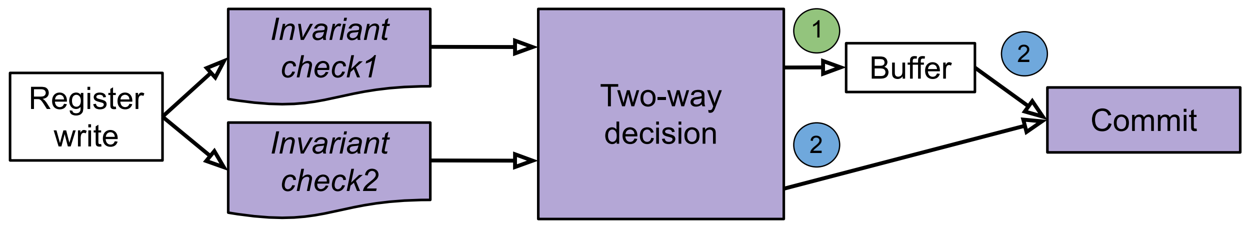


Figure 4.6: Two-way invariant check design.

4.5.2 Viola's Solution

In Viola, we build a two-way invariant using two one-way invariants:

- `sensor on` $\rightarrow$ `indicator on`: indicates that whenever the sensor is on the indicator is on.
- `indicator on` $\rightarrow$ `sensor on`: indicates that whenever the indicator is on the sensor is on.

If both invariants hold, then the two-way invariant holds. It is easy to show that this means that the states of both the sensor and indicator should change together.

Figure 4.6 shows the design of the two-way invariant check. There are four main components. The first and second components are checks for the two aforementioned invariants. Register write parameters are input to the two-way invariant check, which passes them to the one-way invariant checks. These checks will determine if the register write is violating their corresponding one-way invariants and pass their results to the third component, called the *two-way decision*, which decides what action to take based on the results of two invariant checks.

Two-way invariant decision. For the two-way invariant to hold, the sensor and indicator must be either both on or off. If they are both off, the only allowed action is turning both of them on. Similarly, if both of them are on, the only allowed action is turning both off.

Since these include two register writes, one for sensor and one for indicator[1], and our two-way invariant check answers to only one register write at a time, we implement the two-way decision in two steps, as illustrated in Figure 4.6. In the first step, we buffer the first register write changing the state of the first device (either sensor or indicator) and skip emulating the register write (i.e., committing it). In the second step, when the register write for the second device arrives, the two-way decision module passes the two register write parameters to the fourth component, commit module. This module then commits both register writes, i.e., the new register write and the register write that was previously buffered. The module commits both register writes atomically in order to prevent an interruption between the two. We will explain how Viola implements atomic commit of the two register writes in §4.5.3.

When to accept, reject, buffer, or commit register writes? The two-way decision module takes action based on the current state of the devices and the results of two one-way invariant checks. It can take three actions on a single register write: accept, buffer, and commit.

Figure 4.7 illustrates a flowchart for determining the action in the two-way invariant check. Upon a register write, the two-way invariant check passes the parameters to the two one-way invariant checks. If both checks return accept, the two-way invariant check commits the write. If either of them returns reject, it then investigates the buffer and if there is no buffered register write, it stores it in the buffer. Otherwise, if the register write is on the same device with the buffered write, it updates the buffer. And if the write is changing the state of the other device, it commits both the incoming register write and the buffered register write atomically and clears the buffer. Note that in two-way notification, we do not reject any register writes. Any register write that is trying to change the device state is either buffered or committed. If any malware tries to turn on only one of the sensor or indicator while keeping the other off or if it tires to turn one of them off while keeping the

[1]Note that more than one register write might be needed to turn on/off an I/O device. In our discussions, we only consider the last register write that effectively turns on/off the device.

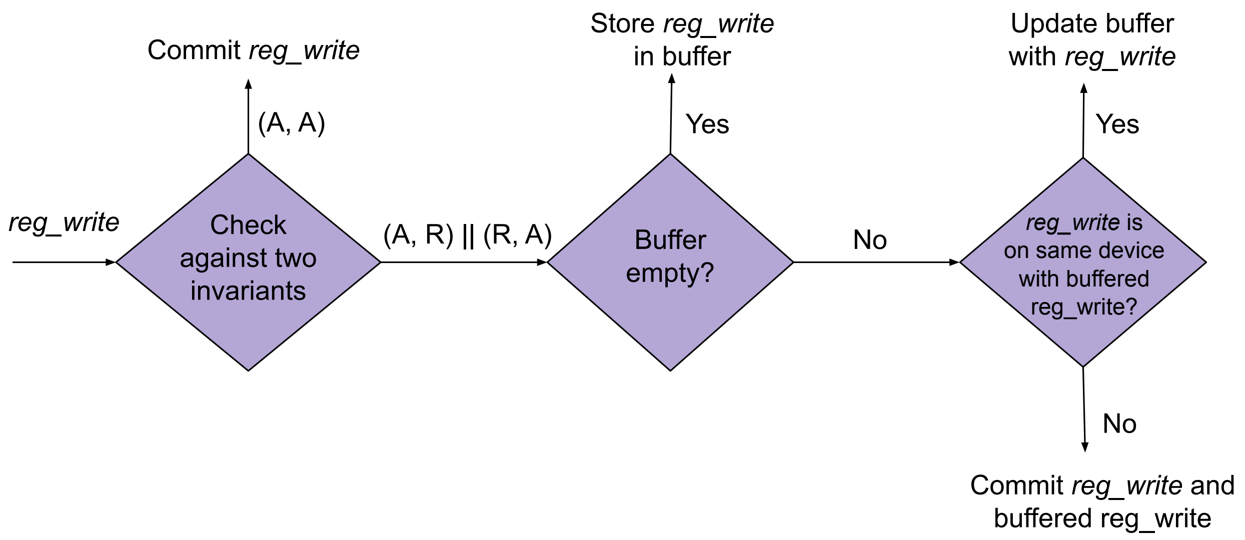

Commit *reg_write*
(A, A)
Store *reg_write* in buffer
Yes
Update buffer with *reg_write*
Yes
reg_write
Check against two invariants
(A, R) || (R, A)
Buffer empty?
No
reg_write is on same device with buffered reg_write?
No
Commit *reg_write* and buffered reg_write

Figure 4.7: Two-way invariant check flowchart.

other on, the register write is only buffered and state of the sensor or indicator will not be changed. Finally, it is easy to show that the two invariant checks never both return reject for the same register write.

Multithreaded implementation of sensor notifications. Viola implements the sensor notifications for one-way notifications in the same thread that activates or deactivates the sensor, i.e., the sensor access thread. Such an implementation faces potential problems for two-way notifications due to the buffering of register writes. More specifically, if the code following a buffered register write depends on the write being executed (e.g., because of a register read), this code segment will not function correctly. We therefore use a multi-threaded implementation of sensor notifications for two-way notifications. When turning on the sensor device, a new notification thread is created. If needed, the sensor access thread or the notification thread can wait for any buffered register write to their corresponding devices to commit before continuing their execution. However, if buffering the register write does not create problems for the following code, these threads do not have to wait, as is the case in our implementation of the two-way notification for camera and vibrator on Nexus 5.

4.5.3 Atomicity on register writes

In order to make sure that both the sensor and indicator reach their final states without interruptions, Viola commits the writes changing the state of sensor and indicator atomically. To do so, we have to consider two aspects:

Concurrency between processes. We need to make sure that other processes that are trying to change the state of the device cannot break the invariant. Therefore, we enforce mutual exclusion among processes as explained in §4.3.1.

Thread preemption. A preemption of the thread executing the two register writes can violate the two-way invariant by executing only one of the writes. We therefore disable preemptions of the thread when it executes the two register writes atomically. To achieve this, we disable (and later enable) hardware interrupts on the corresponding CPU core.

4.6 Implementation

4.6.1 Runtime Monitor

Viola's monitor intercepts writes to registers of sensors and indicators by removing the write permissions from their page table entries. Upon a page fault, it consults with the deployed invariant checks; if the checks pass, the monitor emulates the faulting instructions, and if they fail, the monitor reports an error to the fault handler.

Viola's monitor emulates register write instructions as follows. It maps the register pages into a second set of virtual addresses, used by the monitor only. Upon each page fault, it inspects the CPU registers to determine the value that was going to be written to the faulting address. It then issues an instruction to write this value to the same register using

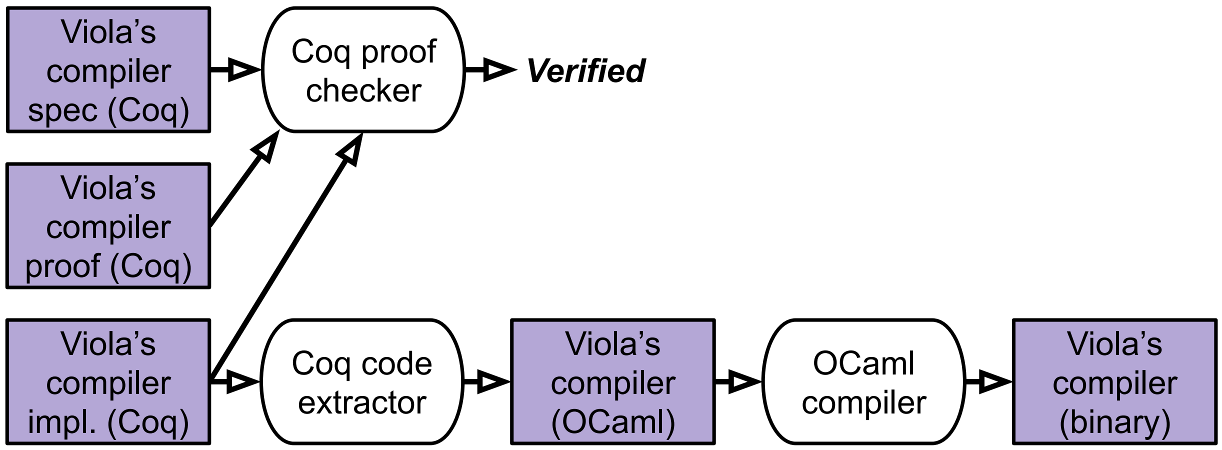

Viola's compiler spec (Coq)
Coq proof checker
Verified
Viola's compiler proof (Coq)
Viola's compiler impl. (Coq)
Coq code extractor
Viola's compiler (OCaml)
OCaml compiler
Viola's compiler (binary)

Figure 4.8: Viola's compiler code in Coq is compiled into OCaml and to machine code. The proof checker verifies the correctness of the proof.

the second virtual address. Finally, it increases the program counter to point to the next instruction and returns.

4.6.2 Verified components

We verify the functional correctness of the compiler. We implement and verify the compiler in Coq, which provides functional programming constructs in addition to logic constructs. The implementation of the invariant compiler translates the invariant logic to Cminor code.

Once we develop and prove the functional correctness of the compilers, we generate their executable machine code. For this, we first use the code extraction facilities in Coq to generate equivalent OCaml code from the source code in Coq and then use the OCaml compiler to generate the machine code. Figure 4.8 illustrates these steps. The same figure also shows that the implementation, the specification, and the proof are passed to the Coq proof checker for checking the correctness of the proof.

4.6.3 Supported Systems and Devices

We test Viola on two smartphones: LG Nexus 5 running Android 5.1.1 (CyanogenMod 12.1) and Samsung Galaxy Nexus running Android 4.2.2 (CyanogenMod 10.1). Since these smartphones do not support a hypervisor mode, we insert Viola's monitor in the Linux kernel. Moreover, to demonstrate the feasibility of a hypervisor-based monitor, we implement Viola on the ODROID XU4 development board as well, which incorporates an Exynos 5422 SoC with virtualization support in its ARM Cortex-A15 and Cortex-A7 processors. On this board, we use the Xen hypervisor (version 4.6.0) and Ubuntu operating system (version 14.04) running on Linux kernel (version 3.10.82). We currently run the operating system in Xen's control domain, i.e., dom0. In this case, we need to deprivilege the dom0 by disallowing the dom0-specific hypercalls in the hypervisor. We have not currently implemented this but we expect it to be trivial.

We test Viola with the microphone and LED on Galaxy Nexus, with the camera, vibrator, and LED on Nexus 5 (two separate sensor-indicator combinations), and (as a proof of concept) with GPIO LEDs on ODROID XU4. The audio chip (TWL6040) and the LED [18] in Galaxy Nexus are accessed through the OMAP4 I^2C bus, for which, we leverage our bus interpreter module. The rest of the devices are memory-mapped. Moreover, as discussed earlier, we currently only support continuous notifications (§4.1.2), which are not ideal for the vibrator. Yet, we use the vibrator in our implementation to demonstrate the applicability of Viola to a diverse set of indicators.

The sensor notification implementation varies in each case. In Galaxy Nexus, we implemented the notification for the microphone in `tinyalsa`, a library used by the audio service in Android. In Nexus 5, we implement the notification for the camera in the camera device driver in the kernel.

4.6.4 Trusted Computing Base

Trusted components for the monitor. The trusted components for the monitor differ for the kernel-based and hypervisor-based implementations. The hardware and the monitor itself are trusted in both implementations. However, in the hypervisor-based monitor, the hypervisor is trusted, but not the kernel (including all its device drivers), which is trusted in the kernel-based implementation. Given that a hypervisor is typically smaller than an operating system kernel, our hypervisor-based implementation provides a smaller TCB. Moreover, while we currently use a commodity hypervisor (i.e., Xen), it is possible to use a smaller hypervisor to reduce the TCB. This is because we do not use the features of the hypervisor needed to run multiple virtual machines in the system, such as scheduling the virtual CPUs between virtual machines. Furthermore, as discussed in §5.3, it is possible to use a verified kernel or hypervisor [122, 125, 142, 113, 88] in the implementation, which will then reduce the TCB to only the hardware. Finally, note that in the kernel-based monitor, only the kernel is trusted, but not the user space including all of Android services and applications.

Trusted components for the verified compilers. The specifications that we develop for the languages' semantics, for the proof, for the devices, and for the buses (i.e., rules), are assumed to be correct. Moreover, the Coq's simple proof checker that verifies the correctness of the proof is trusted but not the tactics used in the construction of the proof. Also, the Coq extraction facilities and the OCaml compiler are trusted. In addition, we leverage an existing assembler in the compiler backend and hence the assembler is trusted. However, if a verified assembler is added to CompCert [126], we can simply use it without requiring any further engineering effort. We can also leverage other solutions for verifying assembly programs [180, 103, 104, 173].

While bugs in any of these trusted components can undermine the functional correctness of our compilers, we believe that Viola provides enhanced trustworthiness compared to the

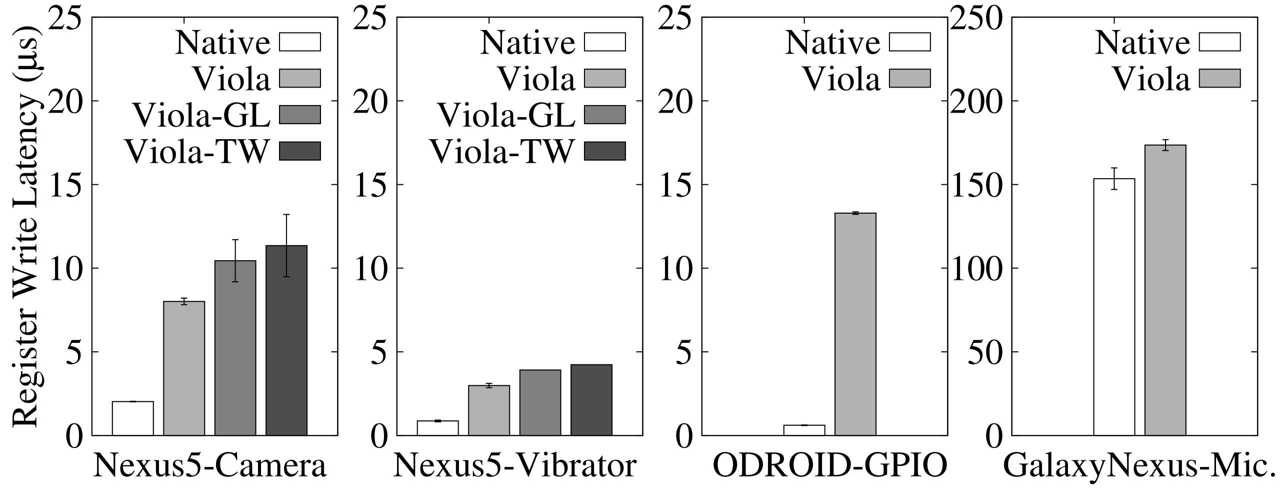


Figure 4.9: Register write latency. Note that the rightmost figure uses a different y-axis range. Also, the third figure is based on a hypervisor-based monitor, whereas the rest are based on a kernel-based monitor.

alternative of (*i*) not using invariant checks or (*ii*) developing the checks manually. In (*i*), the whole operating system and the applications will be trusted. In (*ii*), not only the implementation of the checks can have bugs and need to be trusted, the compiler for the language used, e.g., gcc, must be trusted as well.

4.7 Evaluation

4.7.1 Engineering Effort

For one to use Viola, one must develop the sensor notification, Viola code corresponding to the notification invariant (e.g., Figure 5.5), the specifications for the devices and, if needed, the rules for the peripheral buses. The first two are quite straightforward. However, the last two can be cumbersome depending on the devices and the buses. However, as mentioned in §4.4.2, often partial specifications are enough in Viola, which significantly reduces the required effort for device specifications. Moreover, device specifications are increasingly available from hardware vendors [161]. Finally, writing the specification for a device or the rules for a bus is a one-time engineering effort. The same device specification and especially the same bus rules can then be reused for various sensor notifications.

4.7.2 Performance

Microbenchmarks. We measure the added latency of Viola to a single register write. We measure it for accessing the camera and vibrator registers in Nexus 5, the GPIO registers in ODROID XU4 (with a hypervisor-based monitor), and the microphone registers in Galaxy Nexus.

Figure 4.9 summarizes the results. It shows the average and standard deviation of monitored register write latency in these systems. It demonstrates that the added latency is significant. Part of this latency is due to the page fault exception and part is due to the code running in the fault handler including a large part of the runtime monitor's code (such as the code that inspects the CPU registers) and the invariant checks. Note that in this figure (as well as the rest of the figures in this section), Viola-GL refers to Viola using a global lock for protection against concurrency attacks (§4.3.1), and Viola-TW refers to a two-way notification (§4.5), which always leverages the global lock.

Moreover, the figure shows that the hypervisor-based monitor incurs higher overhead. This is mainly because a trap in the hypervisor incurs a virtualization mode switch (from the kernel mode to the hypervisor mode), whereas a page fault in the kernel does not since both the faulting code and the fault handler are in the kernel mode.

Another observation is that the register write latency for microphone on Galaxy Nexus is much higher than the rest. This is because the microphone registers are accessed through the

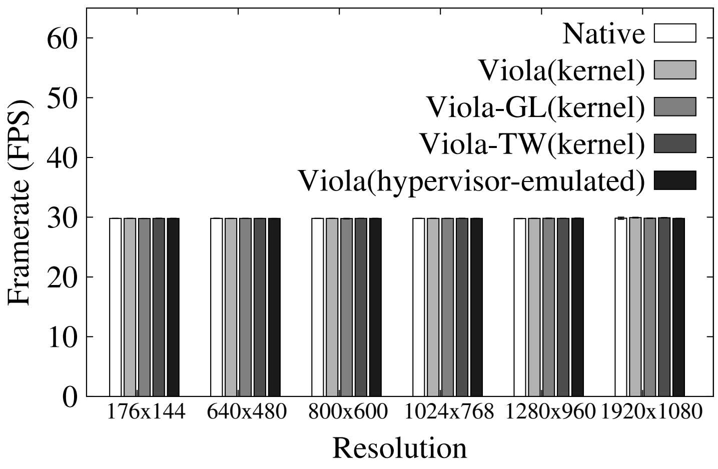


Figure 4.10: Camera's performance in Nexus 5. Note that we emulate the overhead of the hypervisor-based monitor in Nexus 5 (§4.7.2).

I^{2}C peripheral bus, which is not only slower than a memory-mapped access, it also requires multiple writes to the bus adapter registers for a single write to the device register.

We also observe that adding the global mutex lock in the fault handler of the monitor in order to protect against concurrency attacks (§4.3.1) adds overhead to a single register write.

Moreover, we notice that a single register write in a two-way notification experiences a slightly larger latency. In the figure, we show the register write latency both for the sensor (Nexus5-Camera) and the indicator (Nexus5-Vibrator). This additional overhead is due to the additional buffering and commit operations in a two-way notification.

Macrobenchmarks.

First, we measure the performance of the camera on the Nexus 5 in terms of the framerate. We measure the framerate for varying resolutions with and without Viola. For each resolution, we measure the average framerate achieved over a 1000 frames. We discard the first 50 frames to avoid the camera initialization overhead and repeat each experiment three times. Figure 4.10 shows the results. It demonstrates that Viola's overhead on the camera performance is negligible irrespective of the resolution.

The reason for this negligible performance overhead is the infrequency of register writes. To demonstrate this, we measure the number of register writes that are intercepted by Viola when the camera is on. We find that there is one register write interception every 19.3 ms, which is significantly larger than the latency of a single register write (i.e., 2 and 8 μs for native and Viola). The number of register writes are relatively small since they are used mainly for sending control commands to the camera and not for exchanging data. Moreover, in the case of camera, Viola does not intercept writes to all the registers (see discussion on partial specifications (§4.4.2)). Only writes to the registers in the same register page as those monitored by Viola are intercepted. Our experiment shows that this constitute 4% of all register writes.

Moreover, we demonstrate that even the more significant register write latency incurred by a hypervisor-based monitor will not impact the performance of the camera. For this, we emulate the overhead of the hypervisor-based monitor in Nexus 5 by artificially adding a delay of about 6.7 μs to our fault handler since this is the additional latency incurred by the hypervisor as derived from the ODROID XU4 results in Figure 4.9. We then measure the camera framerate and show the results in Figure 4.10. The results demonstrate no performance overhead. This is, similarly, due to the infrequency of register write interceptions.

Second, we measure the performance of the microphone on Galaxy Nexus in terms of the audio rate. We measure the audio rate for a one-minute recording experiment and with varying audio buffering sizes, which determines the audio latency. We repeat each experiment three times. Figure 4.11 shows the results. Similar to camera, we notice that Viola's overhead on the microphone performance is negligible. We also measure the number of intercepted register writes for the microphone and find them to be very small, i.e., only about 20 when starting the capture.

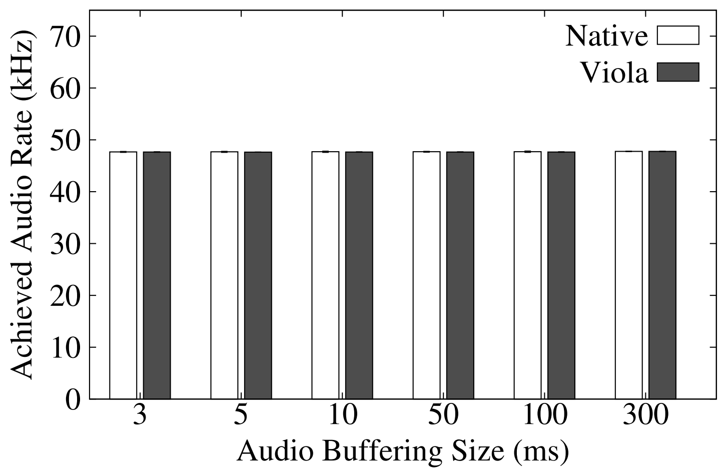


Figure 4.11: Microphone's performance in Galaxy Nexus based on a kernel-based monitor with or without Viola.

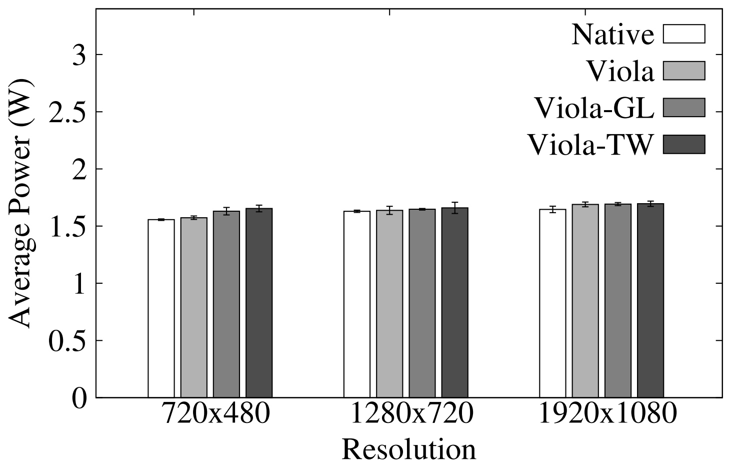


Figure 4.12: Power consumption of Nexus 5 when running a video recording application [1] in the background and with the display off.

4.7.3 Power Consumption

Battery lifetime on mobile systems is limited. Therefore, it is important that Viola does not enhance user's privacy at the cost of reduced battery lifetime. We therefore measure the power consumption of the system with and without monitoring by Viola. We measure the power consumption using the Monsoon Power Monitor [15].

First, we measure the power consumption of Nexus 5 when recording a video. To magnify the relative overhead by Viola, we put the camera application [1] in the background and turn the display off. Figure 4.12 shows the results. It demonstrates that Viola incurs additional power consumption but that the overhead is small (less than 45 mW, 75 mW, and 99 mW for Viola, Viola-GL, and Viola-TW, respectively). The high baseline power consumption of the smartphone is mainly due to the CPU and camera being on.

Second, we measure the power consumption of Galaxy Nexus when using an application [2] to record audio. Similarly, to magnify the relative overhead of Viola, we turn the display off. Our measurements show that, for audio buffering size of 300 ms, the power consumption of the smartphone is about 1.025, and 1.038 W for native and Viola, respectively. The additional overhead is about 13 mW, smaller than the video recording overhead, which might be partly due to different hardware in the two smartphones, and party due to fewer intercepted register writes. Moreover, note that the baseline power consumption is high because of the CPU.

Chapter 5

Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices

Mobile and Internet-of-Things (IoT) devices, such as smartphones, tablets, wearables, voice-activated smart home assistants (e.g., Google Home and Amazon Echo), and wall-mounted cameras, incorporate various sensors, notably cameras and microphones. These sensors can capture extremely sensitive and private information, such as video and audio. There are increasingly important scenarios, where it is essential for these devices to provide assurance about the use (or non-use) of these sensors to their owners or even to a third party. For example, the owner of a home assistant might require assurance that the microphone on the device is not used during a given time of the day. As another example, during a confidential meeting, the host might need assurance that microphones and cameras of the attendees' smartphones remain turned off.

Despite compelling use-cases, there is currently no systematic and secure way to provide hard assurance about the *sensor activities* in mobile and IoT devices. Current practices are ad hoc and crude: At home, the owner of the home assistant might physically unplug the device or merely rely on the microphone disable button/LED on the device. In a confidential meeting, the host may either physically sequester attendees' mobile devices or simply ask them verbally to avoid recording. Such ad hoc measures have important limitations: (1) they are either too restrictive and draconian (e.g., physical unplugging or sequestering), causing inconvenience to the users, or (2) they do not provide any firm guarantees (e.g., relying on a potentially compromised microphone disable button/LED or on a verbal request), thus resulting in undetected policy violations.

To address this issue, this paper presents Ditio, a system for *auditing* sensor activities in mobile and IoT devices. Ditio records sensor activity logs; when needed, an auditor can check these logs for compliance with a given policy. Ditio is designed and built based on the security monitor proving a small Trusted Computing Base (TCB). The hypervisor layer, supported by virtualization hardware, enables Ditio to efficiently record every access to registers of selected sensors without making any modifications to the operating system. TrustZone, on the other hand, enables sealing of logs to guarantee integrity, authenticity, and confidentiality.

While Ditio's implementation is generic and device-agnostic, its log content is sensor-specific. Most log entries correspond to a *write* to, or a *read* from, a specific sensor register. Behavior of the sensor, as a result of these register accesses, can only be understood if adequate information about the interface of the sensor is provided. Therefore, parsing and analyzing the logs can be challenging, cumbersome, and, more importantly, error-prone for the auditor. To address this challenge, we provide a formally verified companion tool, or companion for short. On input of: logs, sensor interface specification, and a policy compliance query, companion analyzes the logs to answer the query.

We implemented Ditio on an ARM's Juno development board that allows programming of both the hypervisor and TrustZone to non-vendors. In addition, for compatibility with older mobile and IoT devices (not equipped with TrustZone and/or virtualization hardware), we provide a secondary design that uses the operating system kernel instead of the aforementioned monitor, and deploy it on a Nexus 5 smartphone.

We report on our experience in deploying Ditio for auditing the activities of a camera on the Juno board and a camera and a microphone on the Nexus 5 smartphone. We show that the auditor can use the companion tool effectively to audit the sensor activity logs. We also show that Ditio does not incur noticeable performance overhead for these sensors. However, it incurs some power consumption increase under heavy use, e.g., by 17% for the Nexus 5 camera.

Ditio is related to our previous work on Viola [147], which provides formally verified sensor notifications. That is, it guarantees that a notification, such as LED light or vibration, is triggered if a privacy-sensitive sensor, such as camera or microphone, is used. Ditio and Viola provide complementary techniques for enforcing policies on the use of sensors. Viola performs runtime policy enforcement: it triggers a notification when the sensor is being used. Ditio enables auditing: it records accesses to sensors and allow future audits. These systems are not equally suitable for use cases involving sensor usage policy enforcement. For example, Viola cannot be reliably used for providing notifications to a third-party due to the lack of a reliable notification channel. On the other hand, Ditio is not suitable for scenarios where a delay in policy violation detection cannot be tolerated. It is, however, possible to use both systems together. For example, in the confidential meeting scenario discussed earlier, Viola can attempt to provide (but cannot guarantee the delivery of) a notification to the host when a sensor is being used. At the same time, Ditio can record logs in case a future audit is deemed necessary.

5.1 Motivation

As mentioned earlier, typical modern mobile and IoT devices (e.g., smartphones, tablets, wearables, wall-mounted cameras, and smart home assistants, such as Google Home and Amazon Echo) include various sensors, notably camera and microphone, which can record private and sensitive information. There are important scenarios where all or some sensor activity in these devices must be restricted or prohibited for privacy reasons. More specifically, in these scenarios, the device must provide assurance about the use or non-use of sensors to either its owner or to a third party.

Assurance for the owner. In the first category of examples, the device needs to provide the aforementioned assurance to its owner. As one example, consider home assistants. These devices incorporate an always-on microphone that listens to users' commands. Such always-on listening causes privacy concerns. Hence, the owner might require assurance that the device remains off during given times.

As another example, consider a confidential meeting. Attendees might need assurance that the microphone and camera on their own smartphones and smartwatches remain off during the meeting.

Assurance for a third party. In the second category of examples, the device needs to provide assurance to a third party.

As one example, reconsider the confidential meeting. The host of the meeting might require assurance that the microphone and camera of the attendees' mobile devices remain off. As another example, consider the same confidential meeting taking place in a rented conference room, e.g., in a hotel, which might be equipped with wall-mounted cameras. In such a setting, the attendees might require assurance from the hotel administrator that the cameras remain off during their meeting.

All aforementioned examples follow a common model: the owner or a third party has a well-defined *policy* about the use (or non-use) of sensors of mobile and IoT devices and needs assurances that this policy is not violated.

Existing methods of satisfying such policies are *ad hoc*. The first category of solutions attempt to provide hard guarantees, but are too restrictive and draconian and hence cause significant inconvenience to the users. For example, the home assistant owner might decide to unplug the device for complete assurance. Or in a confidential meeting, either the attendees do not take their mobile devices with them, or are forced by the host to relinquish them. Similarly, in a rented conference room, the attendees might attempt at unplugging the wall-mounted cameras.

The second category of solutions are easier to implement but fail to provide any hard guarantees. For example, being aware of users' privacy concerns with home assistants, these devices provide a physical button for the user to turn the microphone off, while leaving the rest of the device on (e.g., its speakers). This can be used by the users to disable the microphone when privacy is needed [7]. Doing so might also change the LED color on the device for user's information [7]. However, a compromised device can easily bypass this measure (i.e., fake the LED color and keep the microphone on despite user's press of the button). As another example, the host of the confidential meeting might only verbally ask the attendees to turn their devices off or ask the hotel owner to turn off the wall-mounted cameras. However, he will not be able to detect any potential violations.

In contrast to these existing methods, Ditio provides an *easy-to-use, reactive, and rebust* solution. Ditio's auditing approach makes it easy for users to set up and use it and puts the burden on the device to provide adequate logs when needed. Moreover, Ditio provides strong guarantees due to its small TCB (§5.3) and its use of formal verification methods to enable the auditor to reliably analyze the recorded logs (§5.5).

5.2 Overview

Ditio is a system solution for auditing the sensor activities in mobile and IoT devices. We use the term “sensor activity” to refer to the use or non-use of sensors. More specifically, we use this term to refer to changes in the states of the sensors, e.g., turning them off or on. We also use the term “client” to refer to the mobile or IoT device that contains the sensor of interest.

Depending on the use-case, auditing can be performed by different parties. For example, auditing of sensors in a home assistant is performed by the device owner. Recall that Ditio logs sensor activities on demand and the user is responsible for starting and stopping the logging. For example, the home assistant owner can turn the logging on or off when needed. Or in a confidential meeting, attendees should turn on camera and microphone activity logging before the meeting starts and turn them off after it ends. If logging is stopped earlier than required, it will be detected by the auditor. If logging is not enabled at all, no logs will be available, which is a violation in and of itself.

Note that the device itself should not be used for analyzing the logs for two reasons: (*i*) it is not trusted and (*ii*) it might not have proper user interfaces making it difficult to submit analysis queries.

In the rest of this section, we describe Ditio’s design and workflow, threat model, and TCB.

5.2.1 Design and Workflow

Ditio consists of four components on the client: (*i*) a trusted sensor activity recorder implemented in the hypervisor, (*ii*) trusted authentication and sealing facilities in the TrustZone secure world runtime, (*iii*) an untrusted log store implemented in the normal world operating

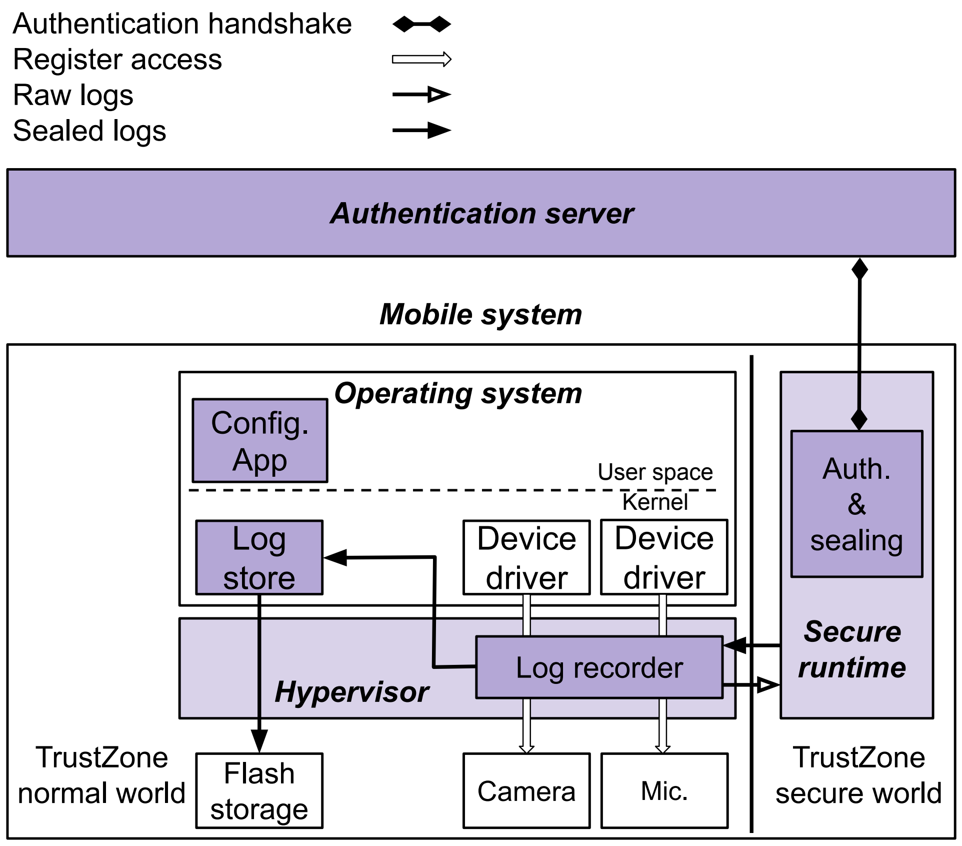


Figure 5.1: Ditio's components on the client (in dark).

system kernel, and (*iv*) an untrusted configuration app in the normal world operating system user space. Ditio also includes an external trusted authentication server and a formally verified companion tool used by the auditor to identify violations of sensor activity policies. Figure 5.1 demonstrates Ditio's design.

Note that we explain the design of Ditio mainly in the context of the security monitor design (§2) based on the hypervisor and TrustZone. The backward-compatible design is mostly similar except that the recorder and the authentication and sealing facilities are all in the operating system kernel communicating using function calls. In the rest of the paper, we will highlight important differences when needed.

Configuration app. Using the configuration app, the user starts and stops customized recording depending on the policy. Customized recording refers to recording accesses to the registers of sensors of interest. The configuration app is Ditio's interface to the user. In practice, the configuration app can be realized differently for different devices. For example, on an Android smartphone, the configuration app is an Android application. On a home assistant, it can be an application running on the user's smartphone or a web portal.

Recoder. Ditio records register accesses since registers are the interface that I/O devices (including sensors) provide to software for programming them. Hence, by analyzing the register accesses performed by the operating system, the auditor can be sure about the state of the sensor at any point in time (e.g., whether the sensor is off or on). The Ditio recorder stores the register access logs in fixed-size log buffers. Once a buffer is full (or if logging finishes, or if the device is about to shut down/reboot), it is committed to flash storage.

Log store. The log store is implemented in the normal world of the client to minimize the TCB size. Once the log buffer is ready to be committed, it needs to be transferred from the hypervisor to the log store. However, to prevent the normal world from tampering with or reading, log buffers are first transferred over the protected channel to the secure world runtime for sealing.

Sealing. Sealing includes encrypting the log buffer, and computing and appending its HMAC. Ditio encrypts the logs to protect its content since it can include sensitive information, such as when the camera was used. It appends the HMAC to protect integrity and authenticity of logs (§5.4.2).

Authentication. We use a symmetric key, i.e., *session key*, to seal the logs. The session key is shared between the client's secure world and a trusted authentication server through an authentication protocol performed in the beginning of the logging period and after every reboot in that period. The goal of this protocol is to: (*i*) inform the authentication server of the client identity, (*ii*) adjust the clock used to timestamp logs, and (*iii*) establish the

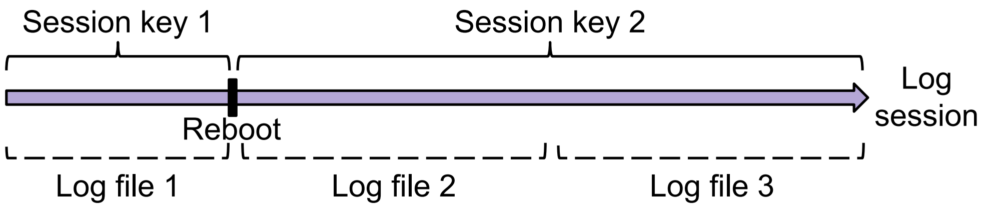


Figure 5.2: Log session example.

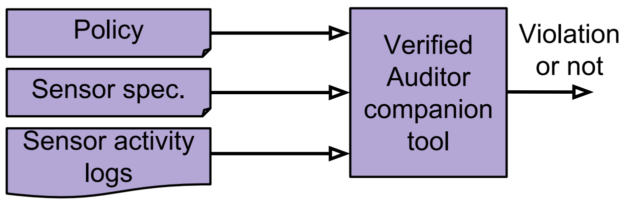


Figure 5.3: Ditio's audit process using the companion tool.

session key. Note that communication between the secure world and authentication server is relayed by the normal world operating system. §5.4 describes this authentication protocol.

Log session. We refer to the logging period as the log session. A log session starts when the configuration app starts recording accesses to registers of sensors of interest and stops when the service stops recording on all registers. A log session can expand over multiple boots of the system. Figure 5.2 illustrates an example. Two session keys are used in the session, one for each boot. Moreover, there are three log files, the first with all the log entries in the first boot and the other two holding part of the entries in the second boot.

There are two important reasons for using multiple log files for a single log session. First, reboots force Ditio to commit the logs. Using a single log file means that Ditio must recover the previously committed log file, append the new logs after the reboot, and recommit the file. This adds to the complexity of the implementation and requires the secure world to re-acquire the previous session key from the authentication server, which complicates the authentication protocol. Second, limited memory space shared between the hypervisor with the normal world and with the secure world also creates a challenge for using a single log file. Therefore, we limit the size of a log buffer in the recorder to be no more than the size of shared memory spaces (two memory pages, i.e., 8 kB, in our prototype).

Connectivity. Ditio requires network connection for two purposes: connection to the authentication server to establish a key, and connection to the same server for sharing the logs. The former is a requirement. The user will not be able to start using Ditio if there is no such connection. The latter is not an immediate requirement. The logs can be shared when connection is established later.

Auditing and the companion tool. Once logging is done, the owner of the client shares its logs with the auditor, if asked. The auditor then asks the authentication server to unseal the logs (§5.4.2). He then uses the companion tool to develop a policy and check the policy against the logs to detect potential violations. Note that this implies that the auditor and the authentication server will have access to the content of the logs and therefore the owner of the client must trust these two entities with the confidentiality of the logs. Figure 5.3 shows the audit process. In addition to the sensor activity logs and the query, the auditor also provides the sensor specification needed to analyze the logs (§5.5).

Note that auditing does not add latency to the log collection process. The third party analyzes the logs offline at an appropriate time. The exact time of auditing depends on the scenario. For example, in a confidential meeting scenario, auditing is performed after the meeting is finished. Or for a home assistant, auditing is done once a week or a month.

Deployment. Several deployment issues are noteworthy. First, as discussed above, Ditio requires modification to various system software layers. We envision it to be deployed on mobile and IoT devices by their vendors. The users can then use the configuration app to interact with it. But users will not be able to "install" Ditio on their devices if not supported by the vendor. However, we note that more tech-savvy users will be able to easily deploy the backward-compatible design by patching and reflashing the operating system. Second, updating Ditio is challenging since it requires updating the hypervisor and secure world runtime. This is a problem with any secure system deployed in these layers. This is one of the reasons we try to minimize our code base in these layers to reduce the frequency of updates in the future. Third, the logs in Ditio only collect information about the usage of sensors. They do not reveal any information that might reveal private information about the vendor of the device, which would otherwise provide a deployment concern for the vendor. Finally, Ditio currently performs its logging in the background. It is, however, conceivable to add some form of notifications on the device to notify the user whenever logging is taking place on the device. We have not implemented this feature.

5.2.2 Threat Model

Ditio protects runtime integrity of the monitor against an attacker that compromises the operating system. This is because these components run in the hypervisor and the secure world, which are isolated from the operating system by hardware. Ditio also protects integrity, authenticity, and confidentiality of sensor activity logs as discussed. This is achieved by: (*i*) using a hardware-bound private key available in the secure world to authenticate the system to the authentication server and establish a shared session key, (*ii*) encrypting the log buffers and computing their HMAC using the session key, and (*iii*) leveraging a non-volatile counter in the secure world to associate log buffers of a single log session in a tamper-evident manner (§5.4).

Ditio cannot protect against Denial-of-Service (DoS) attacks, i.e., it does not guarantee availability of logs to the auditor. Several forms of DoS attacks are possible. First, a malicious operating system can refuse to forward messages between the secure world and authentication server, refuse to store logs, or delete them afterwards. Second, a malicious device (or user) can disable the network interface card, erase logs, or refuse to provide them to the auditor. It is up to the party requesting the audit to deal with such cases. For example, in a smarthome environment, the owner can discard or return an IoT device that fails to provide sensor activity logs repeatedly. Or the confidential meeting, the host might decide to report the attendee.

As mentioned in §5.1, Ditio targets two categories of use cases: one where devices provide assurance for their owners and one that they do so for a third party. The latter (i.e., assurance for a third party) is vulnerable to physical attacks that cannot be protected by Ditio. Such attacks can come in two forms. First, the attacker can tamper with the client's hardware, e.g., by installing additional microphones or cameras. Second, the attacker can introduce an additional, hidden device. In the confidential meeting scenario, a malicious attendee can sneak in a wearable microphone. If needed, protecting against such attacks requires physical defenses, e.g., body search, and hence is beyond the scope of this paper. As related to physical attacks, we note that we assume the user of Ditio knows which devices to target for auditing. For example, in the meeting, the host may audit the smartphones, tablets, or smartwatches of attendees (those devices that are not hidden).

We emphasize that the first category of use cases (providing assurance for the owners) are not easily vulnerable to physical attacks since we assume that the owners are sure of integrity of the hardware in their devices. For example, home assistant, smartwatch, or smartphone owners can trust that no malicious and additional microphone is added to their devices.

Ditio does not protect against side-channel attacks. This has two implications. First, Ditio cannot protect the confidentiality of logs against side-channel attacks. Second, an attacker can try to use other sensors as side-channels to record information of interest. For example, Michalevsky et al. showed that a gyroscope can be used to recognize speech, although the recorded signal frequency and recognition accuracy is low [145]. In this case, it is possible to use a different Ditio policy to also restrict the use of gyroscope in addition to microphone.

The authentication server should be trusted by both the auditor and the user. For example, for the home assistant scenario, the owner of the device can provide the authentication server. Or for the confidential meeting scenario, the host can provide it as long as the attendees trust the host. If not, they can choose a mutually trusted third party server for this purpose.

Finally, while Ditio encrypt all the logs to protect them against unauthorized access, the logs are decrypted and accessed by the auditor. Therefore, the user needs to trust the auditor with access to the content of the logs.

5.2.3 Trusted Computing Base

The secure world runtime and hypervisor in the client are trusted. An attacker who compromises the secure world runtime can tamper with the logs or generate fake ones. An attacker who compromises the hypervisor can bypass logging altogether. They are deployed and locked by the client vendor and are not reprogrammable by users. Therefore, the vendor is also trusted.

We believe that the hypervisor and secure world form a small and reliable TCB on the client. One concern is with security vulnerabilities of commodity hypervisors and secure world runtimes [76, 98]. Although we use commodity hypervisor and secure world runtime in our prototype (i.e., Xen and Open-TEE [16]), more secure hypervisors and runtimes can be used. This is especially the case for the hypervisor since it does not need to support many functionalities needed for running multiple virtual machines in the system. For example, it is possible to use a verified hypervisor or a microkernel [122, 125, 142, 113, 88].

The operating system kernel is not trusted in the design. However, in the backward-compatible design, it hosts the log recorder and the authentication and sealing facilities and is thus trusted. The client hardware is trusted as well. The certification authority (CA), which issues the identity certificate for the client, and the authentication server are both also trusted.

We note that configuration app is not trusted. An attacker can try to disable logging using a compromised configuration app. This will be recorded in the logs and later detected during the auditing phase.

5.3 Recording Sensor Activity Logs

To log sensor activity, Ditio records parameters of read and write accesses to sensor registers of interest. It also records other events needed for the auditing process, such as: (1) start and stop times of a session, (2) timestamps of power-on and power-off events in a session, and (3) the timestamp correction offset (computed vs. a reference time), as discussed in §5.4.1. In this section, we describe how the recorder works.

The recorder is implemented in the hypervisor, which can intercept all sensor register accesses in the operating system using ARM's Stage-2 page tables. This is because all I/O device registers are memory-mapped (i.e., Memory Mapped I/O or MMIO) in ARM's architecture. To record accesses, the hypervisor removes the Stage-2 page table entry's read and write permissions to intercept the operating system's accesses to a given register page. This forces register reads and writes to trap into the hypervisor. The hypervisor then records access parameters, i.e., the target register offset and the value to be written or value read. To obtain these parameters, the recorder inspects the content of the CPU registers in the trap handler and decodes the trapped instruction. It then emulates the register access directly in the hypervisor before returning from the trap. Emulation is done by reading from or writing to the same register through a secondary mapping in the hypervisor. Note the backward-compatible design employs the single level of page tables used by the operating system to force register accesses to trap.

5.3.1 Untrusted Log Store

We designed the log store to be untrusted in order to minimize the TCB. More specifically, we use the existing operating system kernel and file system for implementing the log store to minimize the size and attack surface of the hypervisor. To do this, the hypervisor shares the logs over shared memory pages with the operating system. Note that the log buffers are encrypted and authenticated (using HMAC) in the secure world before they are shared with the operating system kernel (§6.3). Also note that the log store is untrusted in the backward-compatible design as well since the files on the file system are accessible to the user space.

5.3.2 Minimizing Recorder Latency

Sealing the log buffers in the secure world and writing them to flash storage incurs high latency. Therefore, performing these operations in the critical path (i.e., in the trap handler in the hypervisor), would significantly affect performance of sensors. Moreover, high latency might even break the device, e.g., by causing a time-out in the device driver. Indeed, in our original prototype, extra latency in one of our tests with the I^2C register accesses resulted in the driver generating a time-out error.

To address this problem, we use three techniques in Ditio. The first technique is an asynchronous log store. Specifically, the hypervisor stores the parameters of register accesses in memory in a *log buffer*. Once the buffer is full, it asynchronously shares it with the secure world for sealing, and then with the operating system for storage. Log buffers are committed when full or when the system is about to reboot or shut down.

The main drawback of asynchronous commit is that in-memory logs are lost if power is suddenly disconnected, e.g., by sudden removal of the battery or unplugging the device. Fortunately, sudden disconnection of power is detectable during the auditing phase due to inconsistencies in the device power-down and power-on events, i.e., a power-on event in the logs without a preceeding power-off event. Moreover, for many mobile devices, accidental removal of the battery is uncommon and, in some devices, difficult.

The second technique uses lock-free data structures and per-CPU log buffers. Doing so avoids using any locks in the register access fault handler.

The third technique uses dynamically-allocated log buffers and wait queues. Once a log buffer is full, it must be sealed and stored. Instead of waiting for the buffer to be emptied, Ditio adds it to a wait queue and allocates a new log buffer in order to continue recording without delay. Once the queued buffer is committed to storage, memory allocated for it is released. In order to defend against attacks aiming to starve the recorder of memory by refusing to store and hence block release of previous log buffers, Ditio sets an upper bound on the number of concurrently queued buffers. By experimentation, we determined 16 MB to be the upper bound that allows for successful logging of all devices in our prototypes. Moreover, most sensors require fewer buffers than this upper bound. If more outstanding buffers than the upper bound are needed at runtime (e.g., due to unexpected high number of register accesses), the monitor simply stalls register writes until empty buffers are available. In such an unlikely case, the sensor might not function, e.g., camera might stop working, or its performance might degrade, e.g., camera's framerate might drop. However, note that this cannot be leveraged by an attacker to hide sensor activity since no register accesses will be missed from the logs.

5.3.3 Configuring the Recorder

As discussed in §6.3, the configuration app requests the recorder to start or stop logging activities on a given register. To facilitate this, the hypervisor-based recorder provides a hypercall for the normal world operating system to request the start and end of logging on a given register.

The aforementioned hypercall requires the physical address of registers of interest. This leads us to consider how configuration app learns the addresses of registers for various sensors. For this purpose, we use two resources: the *device tree file* to find out the range of physical addresses of all registers of the device, and the device driver or device specification to find out the offset of the registers of interest.

In modern mobile and IoT devices using ARM System-on-Chips (SoCs), physical addresses of register pages of sensors are fixed and declared in a device tree file. Figure 5.4 shows part of the device tree of the camera and audio codec of Nexus 5 used in our prototype, which uses a Qualcomm MSM8974 Snapdragon 800 SoC. As the figure shows, the camera entry shows the physical address of the register pages of camera. The start physical address is `0×fd8c0000` and the size is `0×10000` (i.e., 16 4kB pages). The second part of the device tree shows an audio codec device connected to a SLIMbus. The device tree also specifies the physical address of the two regions of the SLIMbus register pages. The start physical address of the first region is `0×fe12f000` and the size is `0×35000` (i.e., 53 4kB pages) and the start physical address of the second region is `0×fe104000` and the size is `0×20000` (i.e., 32 4kB pages).

Note that in practice, Ditio does not need to monitor all the registers. For example, in case of Nexus 5 camera, logging only one register is enough to infer the on-off state of the device (§5.5). The information about which registers need to be monitored can be easily provided by the device vendors that deploy Ditio. In practice, as mentioned earlier, we identify the right

```
/**** camera ****/
msm-cam@fd8C0000 {
   compatible = "qcom,msm-cam";
   reg = <0xfd8C0000 0x10000>;
   ...
};

...

/**** audio codec ****/
slim@fe12f000 {
   compatible = "qcom,slim-ngd";
   reg = <0xfe12f000 0x35000>,
         <0xfe104000 0x20000>;
   ...

   taiko_codec {
      compatible = "qcom,taiko-slim-pgd";
      ...
   };
};
```

Figure 5.4: Part of the device tree for camera and audio codec of Nexus 5.

registers by inspecting the sensor's device driver or the device specification. For example, it took us less than a day to identify the registers that we needed to monitor for the Nexus 5 camera by inspecting its device driver.

In the backward-compatible design, we need the virtual addresses to which physical addresses are mapped. This is because this design uses the operating system page table for monitoring register accesses, and this table translates kernel virtual addresses to physical addresses. Therefore, this requires a small kernel code modification for runtime discovery of these virtual addresses, given the physical addresses.

Trap-not-log: Protection of register accesses (i.e., forcing them to trap) happens at register page granularity. However, a register page contains many registers, not all of which might be of interest in the auditing phase. Therefore, Ditio avoids logging accesses to other registers in the same page that are not of interest. We implement *trap-not-log* to achieve this: accesses to all the registers in the page are trapped, yet only those of interest are logged. Note that all trapped register accesses are emulated regardless of whether they are logged or not. In §6.8, we show that trap-not-log, while simple to realize, is indeed important in reducing the overhead of Ditio.

Reconfiguration after reboots: Upon each reboot, all changes to the page table entries by the hypervisor are erased. Therefore, it is important for these changes to be re-applied after reboot. We rely on the configuration app to re-initiate monitoring of the same register(s) after each reboot. The configuration app maintains these registers and asks the hypervisor to monitor them after the reboot. One concern is that a malicious user might prevent the configuration app from re-initiating monitoring after the reboot. However, such an attempt will be detected in the auditing phase.

An alternative is to remember monitored registers by storing them in secure storage of the secure world and re-apply the page table protection after reboot. We opted for the previous approach since it simplifies the design by not requiring a large amount of secure storage, which is indeed not trivial to implement [159].

5.4 Sealing the Logs

In Ditio, we *seal* the logs to provide three important guarantees: authenticity, integrity, and confidentiality. The first two guarantees are needed by the auditor. The authenticity guarantee ensures the auditor that the logs were generated by the device of interest during the time period of interest. The integrity guarantees ensures that the logs have not been tampered with since generation. The third guarantee is provided to protect the privacy of the device owner as the logs can contain sensitive information, e.g., the times when the camera is used. With this guarantee, the logs are only readable by the device owner as well as the auditor and the authentication server.

The underlying technique enabling Ditio to provide these guarantees is an *authentication protocol* that allows the client's monitor to not only authenticate itself to a server and adjusts its wall-clock, it also provides a *session key* for the client's monitor that can be used for computing HMAC and for encryption. Below, we first describe this protocol. We will then explain how we provide each of the aforementioned guarantees.

5.4.1 Authentication Protocol

At the start of each log session and after every reboot, the secure world performs an authentication handshake (Auth-Prot) with an authentication server (AuthSrv). As part of it, the client proves its identity to this server, synchronizes its wall clock with it, and establishes a symmetric key (i.e., a session key). In the context of Auth-Prot, we make the following assumptions. First, we assume that the AuthSrv is trusted. This can either be a trusted public entity or a private entity designated for Ditio. Second, we assume that the client has a certificate issued by a well-known and trusted certification authority (CA), e.g., the client vendor, such as Apple, Samsung, Google, and Cisco. This certificate securely binds the client's unique identity to a distinct public key. The public key corresponds to a hardware-bound private key, which is only accessible to the TrustZone secure world in the monitor (and not even the hypervisor). §5.6.1 describes how we use a hardware-unique key in TrustZone's secure world to implement a hardware-bound private key. In our backward-compatible design, we include this key in the operating system image.

The protocol works by exchanging two messages. We describe each message using notation similar to the one used by Needham et al. [155]. That is: $A \rightarrow B:\ C$ means that entity A sends message C to entity B.

$$M \rightarrow \textsf{AuthSrv} : \{Cert\_M, T_M, N_M, \{S_k\}^{Pu_{\textsf{AuthSrv}}}\}^{Pr_M}$$

First, the client (M) sends a message to AuthSrv, which includes M's certificate ($Cert\_M$) and a newly generated symmetric session key (S_k), encrypted with the public key of AuthSrv (Pu_{AuthSrv}). The message also includes a nonce (N_M) and M's timestamp (T_M). The latter can be subsequently used by AuthSrv to order session keys. The message is signed with M's private key (Pr_M).

Upon receiving this message, AuthSrv extracts M's public key Pu_M from $Cert\_M$ and verifies the signature. This process includes $Cert\_M$ expiration and revocation[1] checking. Next, AuthSrv decrypts the message to retrieve S_k. and stores it in a database along with M's certificate.

We note that AuthSrv authenticates contents and origin of the message. Also, since the message includes both a nonce and a timestamp (N_M and T_M), AuthSrv can establish freshness, i.e., detect replay or re-ordering attacks, assuming that M's clock is guaranteed to be monotonically increasing, and that AuthSrv maintains the last valid T_M. However, since M's clock might drift, it might be unrealistic to expect M's and AuthSrv's clocks to be always synchronized. Thus, AuthSrv might not detect message delay attacks. We do not consider this to be a serious issue.

$$\mathsf{AuthSrv} \rightarrow M : \{HMAC_{S_k}(MSG\_1, ID_{S_k}, T_{\mathsf{AuthSrv}}), ID_{S_k}, T_{\mathsf{AuthSrv}}\}$$

Next, AuthSrv replies to M with a message that contains its current timestamp (T_{AuthSrv}) and an ID_{S_k}, which is simply an ID for S_k. Later on, M appends ID_{S_k} to the log file encrypted with S_k. In the audit phase (§5.4.2), AuthSrv uses ID_{S_k} to retrieve S_k. The message also includes an HMAC computed with S_k over T_{AuthSrv}, ID_{S_k} as well as over the entire previous message (MSG_1). HMAC guarantees integrity and origin authenticity of the message; it also (since MSG_1 includes N_M) authenticates AuthSrv to M. T_{AuthSrv} allows M to compute

[1]Revocation checking can be done on-line, e.g., via OCSP, or off-line, e.g., via CRLs. There are well-known tradeoffs in using either approach.

its wall clock offset compared to a reference time. We assume that AuthSrv can provide an accurate timestamp based on an agreed-upon time reference. Note that we do not consider the time taken to send and receive messages between M and AuthSrv. This might result in a clock skew. However, we expect it to be on the order of milliseconds and thus acceptable for anticipated use-cases. To defend against delay attacks, the secure world must terminate the handshake if a response is received after a delay threshold.

Finally, we note that the first message in our protocol has similarity to the well-known X.509 Authentication Procedure [120], but our second message is different.

5.4.2 Audit Phase

When a client is audited, it provides the log files belonging to a session to an auditor along with its certificate. The auditor is responsible for verifying that the certificate corresponds to the client of interest (e.g., by inspecting the device info, or for better assurance, by performing a challenge-response handshake with it).

Once the certificate is verified, the auditor provides the log files and the certificate to AuthSrv. AuthSrv provides two services: (1) verifying integrity and authenticity of the logs, and (2) decrypting them. AuthSrv uses the ID_{S_k} (appended to the log files) to retrieve the corresponding session key for each log file and uses it to verify the HMAC and to decrypt the log files.

Log authenticity: The procedure above guarantees the authenticity of the logs. This is because the session key is only available to the authenticated device's monitor, and hence it is only the monitor that could have generated the HMAC.

Log integrity: The appended HMAC also guarantees the integrity of each log file. However, unfortunately, merely protecting integrity of each log file is not adequate as the attacker can try to delete some log files and violate integrity of the log session. Therefore, we need to make sure that the session's log files are connected to each other in a tamper-evident manner, such that these attacks can be detected. We achieve this by using a *non-volatile counter* in the secure world [73]. The secure world sets the counter to one in the beginning of the log session and re-sets it to zero upon session termination. When preparing a log file, the secure world inserts the counter into the file and increments it. This way, the auditor can always verify that log files start at one and that all the subsequent log files are available. The last log file should contain an event that shows recording on all pages were terminated. In case of per-CPU log buffers discussed in §5.3.2, log files are tagged with counters in the order they are passed to the secure world.

Note that Ditio relies on some form of non-volatile storage in the secure world to maintain this counter value. As described in the TrustZone's specification [24], the secure world provides such a counter (although our development board does not fully implement this as described in §5.6.1 forcing us to emulate it). Moreover, TrustZone-based systems can provide secure storage support [11] but the implementation of that is challenging and missing from processors manufactured by all major SoC vendors [159]. Therefore, we do not leverage this storage to simplify the monitor.

5.5 Formally Verified Companion Tool

The goal of the companion is to assist the auditor in finding policy violations in the logs. The companion has two components: a frontend and a backend. The backend is a formally verified component that makes it easy for the auditor to analyze the device-specific logs while providing formal guarantees on the correctness of the results. The frontend receives the log files, fixes the time stamps, sorts the per-CPU log files, and then passes them to the backend. We next describe the backend in more details.

```
Definition cam_rec :=
    State cam_spec [cam_clk_enable].

Definition cam_query :=
    Query cam_rec start_time end_time.
```

Figure 5.5: The implementation of a policy query in the high-level query language.

As mentioned in the beginning of the chapter, while the recorder itself enjoys a generic device-agnostic design, the content of the logs are specific to the sensors that are being recorded. Each sensor has a unique hardware interface consisting of several registers with different effects on the behavior of the sensor. Moreover, each register is often an aggregation of multiple variables, hence requiring low-level bitwise operations for log analysis. Finally, the behavior of the sensor depends on other components as well, such as its peripheral bus. Auditor's attempt to analyze such low-level logs without additional help can cause misinterpretations of the logs, leading to errors.

We address this problem using a *formally verified companion backend*. As input, the backend receives a policy query written in a high-level language, the sensor activity logs, and the specifications of the sensors represented in the logs. The queries that we support are in the following form: *Was the sensor in a given state between a start time and an end time?* The backend then generates a low-level checker that analyzes the logs to find policy violations.

Figure 5.5 shows an example of a policy query on the Nexus 5 camera written in this language. The language consists of two main programming constructs: *State* and *Query*. Using *State*, we specify the target state of the sensor that is of interest to the auditor. Using *Query*, we generate the query by passing the target state, the start time, and the end time. The first parameter, target state, specifies the state that the auditor is inspecting. In this case, he is looking to find instances of attempts to turn on the camera clock (which is turned on if the camera is turned on). The last two parameters define the time period, during which this query will look for policy violation.

We formally verified correctness of the translation of the checks from the high-level query to assembly. The formal proof provides an important advantage: it enables the auditor to prove correctness of its decision to others. To do this, the auditor can provide the high-level audit query, the low-level checker, and the proof of the correctness of the translation. The audit query is often simple and can be easily inspected by all parties (Figure 5.5). Moreover, if needed, one can check the proof and run the check against the logs to arrive at the same decision as the one by the auditor. This is mainly possible due to the small size of our companion backend: about 500 lines of code for the translator (in Coq) and an interface module to pass input to the generated checks.

To prove correctness of the checker code, we prove that if the sensor specification is correct, the checker that the backend generates is a correct assembly translation of the query provided by the auditor. We perform formal verification using Coq and its proof assistant [20]. The translator is fully implemented in Coq and proved correct using forward simulation [135, 136], similar to other systems [147, 174]. Specifically, our translator translates the code in query language to Cminor, which is an intermediate form of the formally verified C compiler, CompCert [126]. We provide a proof of soundness and completeness of this translation. Translation from Cminor to assembly code is then achieved by CompCert, which is already verified.

Below is the theorem we prove: that the translator correctly preserve the semantics of the original query code while translating it to Cminor:

Theorem *Query_translate_correctness:* `forward_simulation`
`(Query.semantics Qprog) (Cminor.semantics Cprog).`

In order to apply forward simulation, we formalize both the high level query program and low level Cminor program with a series of states (the formalization of Cminor is already provided by CompCert). Programs go from one state to another through these steps. In forward simulation, we particularly prove that (*i*) every start state in the high-level query program is equivalent to the start state of query program in Cminor and (*ii*) every step of the high level query program relates to a sequence of steps in the Cminor program in a way that the starting and ending states in both are equivalent. The second proof is the major part of the overall proof and we formalize it as the following lemma:

Lemma *translate_step:* ∀ `Q1 Q2, Backend.step Q1 Q2` → ∀ `C1, equivalent_states Q1 C1` → ∃ `C2,`
`plus Cminor.step C1 C2` ∧ `equivalent_states Q2 C2.`

In this lemma, Q1 and Q2 are any two states in the high level program with a step between them. C1 and C2 are any two states in the low level Cminor program with one or more steps (`plus Cminor.step`) between them. The lemma shows that if Q1 and C1 are images of each other (equivalent_states), Q2 and C2 are also images of each other. By proving the theorem and in particular the aforementioned lemma, we formally prove that both programs will return the same answer to the query given the same device specifications and same query.

The translator requires the sensor specifications to generate the checks in the companion backend. As noted by others [161], I/O device specifications are increasingly available from vendors. Moreover, Ditio does not require full specifications of a sensor. If the policy is not concerned with a functionality of the device and is only concerned with the device being on and off, the specification can be a few tens of lines of code (§5.7.1).

Finally, some sensors are not directly mapped in the CPU address space. Instead, they are accessed through a peripheral bus, such as I^{2}C. In these cases, Ditio records accesses to the peripheral bus adapter registers. The companion then infers the register accesses of the device from the register accesses of the peripheral bus adapter, similar to the bus Interpreter module in [147]. Similar to the senors themselves, we expect the specification of the bus to be available from their vendors. However, if not, it is possible for the system designer to develop the required partial specification. To demonstrate this, we have developed the specification of the I^2C bus for the Juno board, which took one of the authors a few days to do.

5.6 Implementation

We implement the full Ditio prototype on ARM Juno r0 development board. The board incorporates the ARM big.LITTLE architecture, with the big cluster of Cortex-A57 CPUs and the LITTLE cluster of Cortex-A53 CPUs, both of which use the ARMv8-A 64-bit instruction set. To the best of our knowledge, ARM Juno development boards (r0, r1, and r2) are the only devices that allow programming of the hypervisor and the TrustZone secure world to non-vendors. Note that we run the operating system in Xen's Dom0 in our prototype. Xen provides a command interface for Dom0 to manage the system, e.g., launch and kill new virtual machines. Since such functionality is not needed in Ditio, this provileged interface of Dom0 must be disabled, although we have not done so in our prototype. We also implement a backward-compatible version of Ditio on a Nexus 5 smartphone since we cannot access the hypervisor and TrustZone's secure world on commodity devices (§2).

We support and test different sensors in our prototypes. On the Juno board, we test a USB-based camera (Logitech C270 HD 720p). On the Nexus 5, we test the memory-mapped backfacing camera and the SLIMbus-based microphone. As mentioned in §5.3.3, we deploy trap-not-log for the Nexus 5 camera. For the other two devices in our prototype, we simply log all the register accesses to their corresponding peripheral buses. While we only support camera and microphone in our current prototype (since they are some of the most privacy-sensitive sensors), we believe that Ditio can be easily applied to other sensors as well, e.g., GPS.

In our Juno board prototype, we use Xen for the hypervisor (version 4.8), OpenEmbedded operating system [17] (version 16.04) running on Linux kernel (version 4.6) for the normal world operating system, and Open-TEE [16] (version 16.04) secure operating system for our secure world runtime. We use the mbed TLS library [13] to perform the Ditio's cryptographic functions in the secure world (for the hybrid design) or in the kernel (for the backward-compatible design). The library provides support for AES encryption, RSA encryption and signing, and HMAC. We use the same library in the authentication server. In our Nexus 5 prototype, we use the Android operating system (CyanogenMod version 12.1) running on the Linux kernel (version 3.4).

5.6.1 Juno Board's Specifics

Three issues about the Juno development board are noteworthy.

First, TrustZone specification [24] envisions a "statistically unique secret key" on the SoC. Juno development board's implementation of this key is a 128-bit hardware unique key stored in "One Time Programmable (OTP) or eFuse memory" [73]. As mentioned in §5.4.1, Ditio uses a hardware-bound private key for digital signature. We use a 2048-bit RSA key hard-coded in the secure world image. For complete secrecy, the RSA key must be encrypted with the hardware unique key and then hard-coded in the image. We note that while the Juno development board supports the hardware unique key, some other implementations of TrustZone do not. Ditio relies on the availability of this key to implement its hardware-bound key.

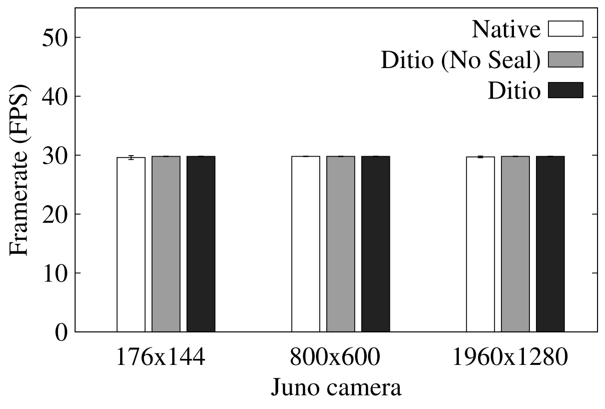
Second, the session key in Ditio is a 128-bit AES key generated by the client. The Juno development board provides a trusted entropy source to generate the session key [73].

Third, the TrustZone specification provisions a 32-bit non-volatile counter [24], needed in Ditio to connect the log files in a tamper-evident manner (§5.4.2). This counter can be used to number and chain more than 4 billion log files. We believe that this number is large enough and that we will not face counter overflow problems in any practical scenarios. In the unlikely case that the counter overflows, the secure world can finish the logging session and starts a new one. The user will then need to provide the logs for both sessions to the auditor.

Following the specification, the Juno board does provision a non-volatile counter in the secure world, however, according to the board documentation [73] and based on our experiments, the value of the counter is always fixed at 31. Therefore, we emulate this counter in our prototype. Moreover, as mentioned in §6.3, Ditio can use the secure storage (i.e., storage space protected by the secure world) for maintaining the counter, an option we have avoided to simplify the monitor (§5.4.2).

5.7 Evaluation

In this section, we experimentally measure the ease of use and overhead of Ditio. We emphasize that in most use cases of Ditio, sensors are not actively used, hence Ditio will incur minimal recording overhead. For example, in the home assistant scenario, the device needs to keep the microphone off. In this case, the logs might simply contain a few register writes showing the microphone was turned off. However, in the experiments, we assume the worst: that is, we assume that logging is active while sensors are heavily used. We perform our measurements in such scenarios. Therefore, our results represent an upper bound on the performance overhead of logging sensor activities.



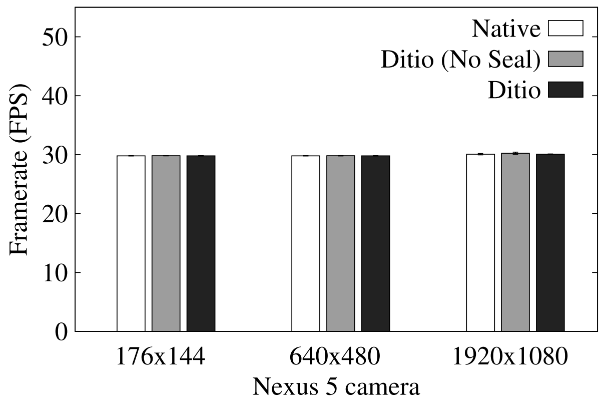


Figure 5.6: Juno board and Nexus 5 camera performance.

5.7.1 Use Cases

To demonstrate the feasibility of using Ditio in practice, we deployed and tested one of the aforementioned use cases: a confidential meeting where no video recording is allowed. More specifically, in this case, we audit the camera on the Nexus 5 smartphone to make sure that it has not been used for a given period of time, i.e., during the meeting. In the experiment, we do use the camera during the designated time period. We then use the companion tool to reliably detect it.

The most important question to answer about this experiment is the ease of use of Ditio. Our experience was that the most time-consuming part of using Ditio was developing the sensor interface specification. Fortunately, the specification is partial; that is, it only captures the

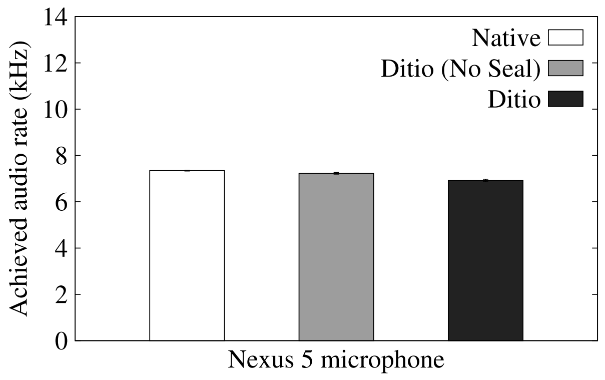


Figure 5.7: Nexus 5 microphone performance.

registers of interest to the event that needs to be detected. As a result, the specification for the camera on Nexus 5 is 13 lines of Coq code. It took one of the authors only a few days to implement this. Note that we expect these specifications to be provided by sensor vendors (§5.5). This will then significantly reduce the engineering effort needed to deploy Ditio.

5.7.2 Sensor Performance

We measure the performance of different sensors when being recorded by Ditio and show that the performance overhead is negligible. For every sensor, we measure the performance of the sensor when used natively, and when used while being recorded. We show the Ditio's overhead with and without sealing in the secure world (the latter marked as "No Seal" in the figures), to quantify the effect of sealing on performance. Moreover, for Nexus 5 camera experiments, we also show the results with trap-not-log disabled to demonstrate the effectiveness of this technique (§5.3.3). Every reported performance number is an average over three runs. For every average number, we also include the standard deviation using error bars in the figures. Note that we do not run any specific applications in the system while evaluating Ditio nor do we pin any of our software components to any cores in the device. In any experiment, we reboot the system and launch our applications for testing such as camera or a recorder application.

Camera performance We measure the performance of the camera by measuring the rate at which it can capture frames, i.e., framerate. We configure the camera to produce the frames at its highest framerate possible (i.e., 30), run the camera for about 1 minute, and measure the rate. We ignore the first 50 frames to remove the effect of camera initialization on the results. Figure 5.6 shows the results for the camera on both the Juno board and Nexus 5 smartphone for different resolutions. We use the IP Webcam [3] application on Nexus 5 and a simple frame-capture program on the Juno board. The results demonstrate that Ditio does not add any noticeable overhead to the performance of the camera.

Microphone performance We measure the performance of the microphone on Nexus 5. Note that we have not implemented support for microphone on our Juno board (§6.6).

For microphone's performance, we measure the achieved audio rate when capturing 60 consecutive 1-second audio segments. We use the Nexus 5 built-in Sound Recorder application in this experiment. Capturing a segment of any size only requires writing to a handful of registers for configuration of the microphone settings and for starting and stopping the capture. Such few register accesses means that Ditio does not add any noticeable overhead. Hence, we use 60 1-second segments rather than a single 60-second segment in order to stress Ditio.

Figure 5.7 shows the results. The results demonstrate that Ditio's affect on the microphone performance is small.

5.7.3 Power Consumption

Although Ditio does not incur noticeable performance overhead to the sensor itself, it increases the power consumption in the system due to the increased computation.

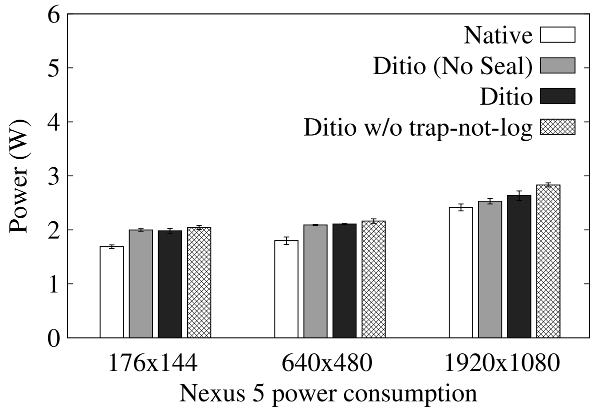


Figure 5.8: Power consumption of using the Nexus 5 camera.

We measure the power consumption for Nexus 5 experiments using a Monsoon power monitor [15]. We focus on camera since it stresses Ditio. Microphone incurs negligible overhead (within the margin of error in our measurement setup). Figures 5.8 shows the results. It shows that logging by Ditio (when camera is actively used) increases the power consumption of the device by at most 17% (for the 176×144 resolution).

The same figure also shows the effectiveness of trap-not-log in achieving acceptable power consumption. More specifically, we repeat the experiment but disable the trap-not-log technique, resulting in all register accesses in the corresponding register page to be logged. In this case, the power consumption can further increase by up to 8% (for the 1920×1080 resolution). However, we note that disabling trap-not-log does not affect the camera performance and it still achieves a close-to-native framerate (about 30 FPS).

5.7.4 Other Results

Log size: We measure the log size generated in each of the experiments in §5.7.2. Our results show the log size to be about 10 MB/min for the camera on the Juno board, and 16 kB/min and 8 kB/min for camera and microphone on Nexus 5.

Similar to power consumption experiments, to demonstrate the importance of trap-not-log, we disable it for the Nexus 5 camera and redo the log size experiments. In this case, Ditio generates about 84 MB/min of logs, which is a significant increase compared to 16 kB/min with trap-not-log. This also explains the large size of logs for Juno board's camera, for which we have not implemented trap-not-log.

Authentication Latency: We measure the authentication handshake round trip time. We use the Juno board for this experiment as the round trip time is affected by the delay incurred by switches between the operating system, hypervisor, and secure world. We host the authentication server on a remote server (not on the university campus where the board is). The network Round Trip Time (RTT) between the board and the server is about 38 ms on average. In this setup, we measure the authentication handshake round trip time to be about 86 ms. We believe this will not incur user-perceivable latency (in the logging initialization in the configuration app triggered by the user). We note that sharing the logs might take a long time but that can be done offline.

Chapter 6

Tabellion: System Support for Secure Legal Contracts

Forming contracts electronically is desirable in many transactions, including real estate sales and leases, venture capital investments, and work for hire, due to its significant convenience compared to traditional methods. Not surprisingly, the global market for electronic signatures and contracts is predicted to grow significantly in the next couple of years. One report estimates the market to grow to \$4.01 billion by 2023 from \$844.7 million in 2017 [38]. Another report estimates the market to grow to \$3.44 billion by 2022 from \$517 million in 2015 [51]. The unfortunate COVID-19 outbreak in 2020 and the resulting social distancing approach deployed to combat it has further accelerated the use of electronic signatures and contracts [167].

As a result of this growth, many electronic contract platforms have emerged [48, 61, 42, 60, 56, 52, 49]. While these platforms are convenient to use, unfortunately, they have an important shortcoming: they do not provide strong evidence that a contract has been legally and validly created (as defined by the law of contracts). More specifically, they do not provide strong evidence that the signatures are authentic, that there was mutual assent, and that the parties had an opportunity to read the contract. As an example, in a recent US court case [33], the court was unconvinced that an electronic signature performed with DocuSign [48] was adequate as it could be manipulated or forged with ease.

In this paper, we present a system solution to provide *strong evidence (i.e., hard-to-fabricate and hard-to-refute evidence)* for the legal and valid formation of a contract on mobile devices. Our system, called Tabellion[1], leverages the security monitor on mobile devices and an SGX enclave in a centralized server. Doing so, however, raises four research questions that we answer in this paper.

Q1. How can the contract platform provide strong evidence for all the requirements of a legal contract? We answer this question in three steps. First, we introduce four secure primitives, three for client devices (secure photo, secure timestamp, and secure screenshot) and one for the centralized server (secure notarization). These primitives can be used to generate strong evidence for a legal and valid contract. Second, we introduce a secure contract protocol to use these primitives to form a contract. Finally, we introduce self-evident contracts, which contain all the required evidence for the legal and valid formation of the contract. That is, each user, and if needed, the court or an adjudicator, can independently verify the contract compliance with applicable law requirements.

Q2. Can the aforementioned primitives be realized securely, i.e., with a small TCB? Having a small TCB for the secure primitives is important as it makes them less prone to software bugs (which can get exploited by attackers), and hence makes the evidence they help generate stronger. Moreover, a smaller code base can be easily inspected for safety and even certified. We show that these seemingly complex primitives can be implemented with ∼1,000 LoC (out of ∼15,000 LoC that we developed in our prototype). To achieve this, we introduce several

[1]Merriam-Webster dictionary defines tabellion as (1) "a scrivener under the Roman Empire with some notarial powers," and (2) "an official scribe or notary public especially in England and New England in the 17th and 18th centuries."

novel solutions including a solution to secure the camera photo buffer, a delay-resistant Network Time Protocol (NTP), and a solution to secure the framebuffer.

Q3. Given that a contract platform provides complex functionalities, e.g., contract viewing, contract submission, and negotiations, can a fully functional platform be realized using the aforementioned primitives and protocol? We answer this question positively and build a fully functional contract platform on top of these primitives and protocol. We discuss how we address several challenges in realizing the required functionality without adding any more trusted code. Indeed, we show that Tabellion's design enables us to add a capability, called *negotiation integrity tracking*, that no existing platform supports.

Q4. Does Tabellion provide strong protection against attacks on a legal contract? We answer this question positively in two parts. First, we define the set of possible attacks on a contract platform including repudiation attacks, impersonation attacks, and confusion attacks. Second, we evaluate the security of Tabellion using a detailed security analysis and show that Tabellion can provide a strong defense against these attacks.

We design and build Tabellion for a mobile-first world where contracts are executed on smartphones and tablets. It is becoming more common to use mobile devices to sign contracts such as mortgages, vehicle leases, and bank loans [165, 152, 166, 99]. Indeed, signing contracts on mobile devices is believed to be the "the future of loans and mortgages" according to a recent study [152]. We leverage the latest mobile research technologies, including the use of TEE and secure biometric sensors now available on modern devices.

We implement Tabellion on a HiKey development board as it allows us to program the TEE (based on TrustZone and virtualization hardware). Note that in this paper, we refer to the term trusted execution environment (TEE) and that is the security monitor we introduced in §2. While necessary for performance measurements and security analysis, we cannot effectively use this board for a user study and energy measurements. Therefore, we build a second prototype of Tabellion on real mobile devices for that part of the evaluation. Since the TEE on these devices is not yet programmable by non-vendors, we emulate the secure primitives in Tabellion's application.

We extensively evaluate Tabellion. We measure the time it takes to carry out various steps of the contract. We show that the execution time of these operations in Tabellion is in the order of several seconds (20 to 35 seconds for very large contracts), which is small enough for a good user experience. We also show that using Tabellion does not consume a noticeable amount of energy. Finally, we evaluate the usability of Tabellion with a 30-person user study. We show that, compared to DocuSign (the state-of-the-art electronic contract platform today), Tabellion provides slightly better convenience (for contract viewing and signing), readability, and understanding of the contract. It also enables the users to read and sign the contracts slightly faster. This demonstrates that improved security in Tabellion does not come at the cost of usability.

6.1 Background

The law of contracts enumerates several requirements for the correct formation of a contract between an offeror and an offeree [92]. Some requirements are beyond the scope of Tabellion as they are concerned with the content of the contract or with the circumstances of the parties involved. For example, the law of contracts requires *consideration*, which states that the contract must represent "bargained for" exchange by both sides of a contract [92]. As another example, the law requires that the parties have not signed the contract under duress and that the parties have legal capacity (e.g., they are of legal age at the time of assent) [92].

There are, however, key requirements that are related to the contract platform. One key requirement in the case of contracts that are required to be in writing and signed is *signature* *attribution*, i.e., that a signature is an authentic signature of the party being charged. Another key requirement is *mutual assent*, which requires that the two parties agree to the same contract. Mutual assent has clear conditions in the law: (*i*) there is an offer from the offeror, (*ii*) there is an acceptance by the offeree, (*iii*) there is no revocation of the offer from the offeror before the acceptance by the offeree, and (*iv*) there is no rejection or counter-offer from the offeree before acceptance [92]. In case of a counter-offer by the offeree, this counter-offer is considered a fresh offer, which may be accepted or rejected by the other party.

Another requirement is that a party has had an opportunity to read a contract. However, it is not a requirement that a party has actually read the contract. In the case of *O'Connor v. Uber Technologies, Inc.* [31], the plaintiffs asserted that there was no valid agreement because it was displayed on "*a tiny iPhone screen when most drivers are about to go on-duty and start work.*" The court rejected that argument because "*for the purposes of contract formation, it is essentially irrelevant whether a party actually reads the contract or not, so long as the individual had a legitimate opportunity to review it.*"

Electronic contracts. The Uniform Electronic Transactions Act (UETA) and the Electronic Signatures in Global and National Commerce (ESIGN) Act permit the use of electronic signatures and contracts. ESIGN states that: "*a signature, contract, or other record relating to such transaction [any transaction in or affecting interstate or foreign commerce] may not be denied legal effect, validity, or enforceability solely because it is in electronic form*" [23]. This means the courts would not reject a contract simply on the basis of it being signed electronically. However, it is up to the contract platform designer to provide a secure solution—one that can be effectively defended in courts. For example, simply printing a contract, signing it using a wet signature, and scanning it does not provide strong evidence for attribution as it is easy to copy/paste (or even forge) the scanned signature. While no platform can guarantee that its contracts will be legally valid with certainty, a platform can increase the odds of success by providing strong and secure evidence for the correctness of its contracts.

Legal cases on electronic contracts. We next discuss a few legal cases to demonstrate the shortcomings of existing electronic contract platforms. First, existing platforms do not provide strong evidence for signature attribution. For example, in *In re Mayfield*, a recent California bankruptcy case [33], the court was unconvinced that an electronic signature created using DocuSign was adequate as it could be manipulated or forged with ease, stating: "*This brings the court to another important problem with Counsel's arguments: they do not address the ease with which a DocuSign affixation can be manipulated or forged. The UST [United States Trustee] asks what happens when a debtor denies signing a document and claims his spouse, child, or roommate had access to his computer and could have clicked on the 'Sign Here' button.*"[2]

Second, existing platforms have failed to provide evidence that the parties agreed to the same contract. For example, in the case of *Adams v. Quicksilver, Inc.* [41, 28], the plaintiff challenged the validity of an arbitration agreement: "*The system provided no audit trail for the signing process, though, so it couldn't be determined when the agreement was signed.*"

Third, existing platforms do not provide evidence that the parties had an opportunity to read the contract. This is evident in the case of *Labajo v. Best Buy Stores* [140, 27]: "*When Christina [the plaintiff] accepted the free subscription, she signed an electronic signature pad at Best Buy [the defendant]. Christina claimed that there was no disclosure telling her she would be charged for the magazine. But whether or not there was disclosure doesn't matter. What matters is the fact that Best Buy couldn't prove that she saw and approved the disclosure.*"

[2]We note that in this case, the court was unwilling to accept software-generated electronic signatures as substitutes for wet signatures due to the California bankruptcy court's local rules of practice. As mentioned, the ESIGN Act provides that a signature cannot be rejected solely because it is in electronic form. However, that provision of the ESIGN Act does not apply to "*court orders or notices, or official court documents.*" [23]

6.2 Attacks on Legal Contracts

We define potential attacks on a legal contract system by either a malicious offeror or a malicious offeree.

Repudiation attack[3]. In this attack, either the offeror or offeree denies having agreed to the contract (when in fact they did). This attack can come in three different forms. First, one party may deny having legally signed[4] the contract. Second, an attacker may deny mutual assent. That is, the parties might disagree on the terms of the contract or on the version of the contract they signed as a result of negotiations (i.e., counter-offers followed by revisions). Moreover, the offeror may claim to have revoked the offer before acceptance by the offeree or the offeree may claim to have rejected or countered it before acceptance. Third, the offeree may claim that they were not given an opportunity to read the contract.

Impersonation attack. In this attack, one party attempts to impersonate someone else and sign a contract on their behalf. There are two forms of this attack. First, the attacker spoofs the victim's authentication on the victim's device. Second, attacker spoofs the victim's identity.

Confusion attack. In this attack, the offeror attempts to fool the offeree into legally signing a contract different from what the offeree thinks they are legally signing. To perform this attack, an attacker needs to target and/or compromise the contract viewer on the offeree's device in order to misrepresent the contract to the offeree. In one special form of this attack, called the Dalì attack [83, 84, 157], the attacker submits a file for the contract in an interpreted document format (e.g., PDF) with dynamic content, which shows different content on the offeror's and offeree's devices.

[3]This is different from the concept of repudiation in the law of contracts, which refers to actions demonstrating that one party to a contract refuses to perform a duty or obligation owed to the other party.

[4]There are two types of signatures in our context. We use "legal signature" or "electronic signature" to refer to a user's assent to a contract, and "cryptographic signature" or "digital signature" to refer to a computer-generated hash that is signed by a private key.

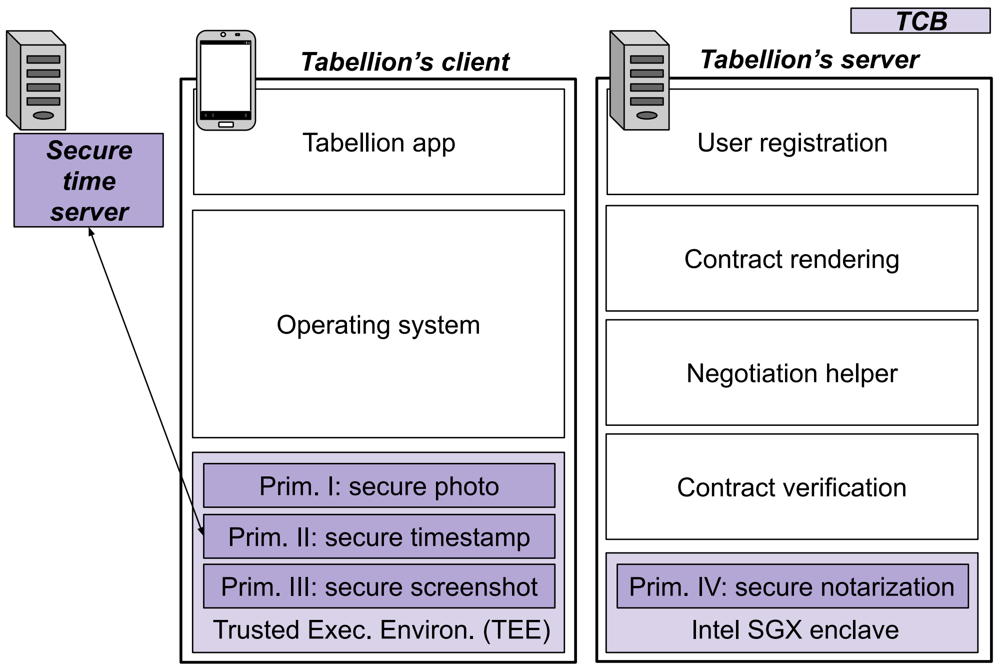


Figure 6.1: The client and server in Tabellion.

6.3 Tabellion: Principles and Design

We present Tabellion, a legal contract platform that provides strong evidence for the legal formation of a contract. Note that while we focus on two-party contracts for clarity, Tabellion can similarly handle multi-party contracts. Tabellion comprises two components: a client that runs on the mobile devices of the offeror and the offeree, and a centralized server that mediates the contract formation. Figure 6.1 illustrates these two components. Each incorporates some code in their TCB to implement the required secure primitives. Each also incorporates a large amount of untrusted code to implement the functionality needed to form a contract. In this section, we introduce the primitives (§6.3.1), discuss the contract formation protocol (§6.3.2), the resulting self-evident contract (§6.3.3), and the contract verification process (§6.3.4).

6.3.1 Secure Primitives

Based on the requirements of a valid and legal contract, we define a set of secure primitives to generate strong evidence for the contract.

Primitive I. Tamper-proof camera-captured photo (i.e., secure photo). Signature attribution in written agreements requires evidence of the identity of the signatory. Photos taken of the user can provide such evidence. Indeed, several existing identity-based systems, such as Voatz [62], capture photos and videos of the user and use them for identification. We define photos as the main primitive since videos are simply a collection of photos.

The key property of this primitive is that the photo must be captured by the camera hardware as opposed to being fabricated by software. The other property of this photo is that the photo must be tamper-proof after capture. We achieve these by reading the photo directly from the camera in the TEE and by cryptographically signing the photo. For the latter, we use a per-user per-device private key, which is generated in the TEE once the user registers with Tabellion on a mobile device (discussed in §6.3.2).

Primitive II. Tamper-proof global timestamp (i.e., secure ti- mestamp). The mutual assent requirement of the law of contracts requires evidence of both an offer and an acceptance of that offer. Moreover, one needs to show that there was no revocation of the offer from the offeror before the acceptance by the offeree and that there was no rejection or counter-offer from the offeree before acceptance. To achieve this, we require the client devices to be able to attach a global timestamp, e.g., a timestamp with respect to a global clock, to each action (i.e., offer, acceptance, revocation, rejection, and counter-offer). The timestamps must be tamper-proof. That is, an attacker should not be able to spoof or modify them. We achieve this using a novel clock synchronization protocol that allows the TEE on the device to securely synchronize its clock with a trusted time server.

Primitive III. Tamper-proof user-confirmed screenshot (i.e., secure screenshot).

The requirement of having the opportunity to read a contract in the law of contracts requires evidence of the contract having been presented to the offeree. We use screenshots of the contents displayed on the client device to achieve this goal. However, not all content displayed on the device is seen by the user. Therefore, we ensure that the user *confirms* seeing the content captured in the screenshot. We also ensure that the screenshots are tamper-proof.

To achieve these, we ask the user to authenticate with the system in order to confirm seeing the displayed content. Upon successful authentication, we use the aforementioned per-user per-device private key to cryptographically sign the screenshot. Note that it is critical that the acts of seeing the content on the display and providing authentication are atomic. If not, an attacker can show some content to the user but have the user unknowingly confirm seeing a different content.

Different authentication solutions can be used. We use biometric authentication, e.g., fingerprint authentication, as it is easy to use, is available on most modern mobile devices, and has high accuracy [25, 57].

Primitive IV. Secure notarization of the contract. This primitive securely connects all the evidence in a contract together so that the evidence cannot be maliciously deleted or reused for another contract, and so that new evidence cannot be added to a contract after it is finalized. We achieve these goals with a secure enclave in Tabellion's server, which acts as a notary by binding all the evidence together and cryptographically signing them.

6.3.2 Contract Formation Protocol

Figure 6.2 illustrates the four steps of the protocol.

Step I: Registration. Both the offeror and offeree first register with Tabellion. In this step, Tabellion's client uses Primitive I to take a photo of the user. It then sends the cryptographically signed photo to the server. The client also sends some additional information to the server, which is needed for self-evident contracts (§6.3.3) including the device TEE certificate and the measurement of the TEE code (which is the certified hash of the TCB code).

The user needs to register with Tabellion once upon installation on a new device. During registration, the client TEE creates a *per-user per-device key pair*. It uses the private key of this pair to sign the user's photo (Primitive I) and uses the same key later to sign the screenshots captured of the content of the contract (Primitive III). It also sends the public key to Tabellion's server for verification.

The TEE code uses biometric-based authentication for the use of the key. That is, the user has to use their biometric (e.g., fingerprint) to confirm the securely captured photo and/or screenshot so that they are signed by the TEE. This strongly binds the user's photo and the confirmed screenshots to each other. That is, if a screenshot and a photo are signed with the same per-user per-device private key, one can conclude that the person in the photo confirmed and signed the contract in the screenshots (see §6.7.2 for a discussion of impersonation attacks and Tabellion's solutions).

Note that, instead of using a separate registration phase, it is possible to securely capture the user's photo for each contract or even for each page in the contract that the user confirms. However, this approach would impose a usability burden.

Step II: mutual identification. When the offeror requests Tabellion to initiate the process of forming a contract with the offeree, Tabellion asks the two parties to confirm each other's identities. To initiate a contract, the offeror names the offeree using a unique identifier, e.g., an email address registered with Tabellion. At this point, Tabellion asks the

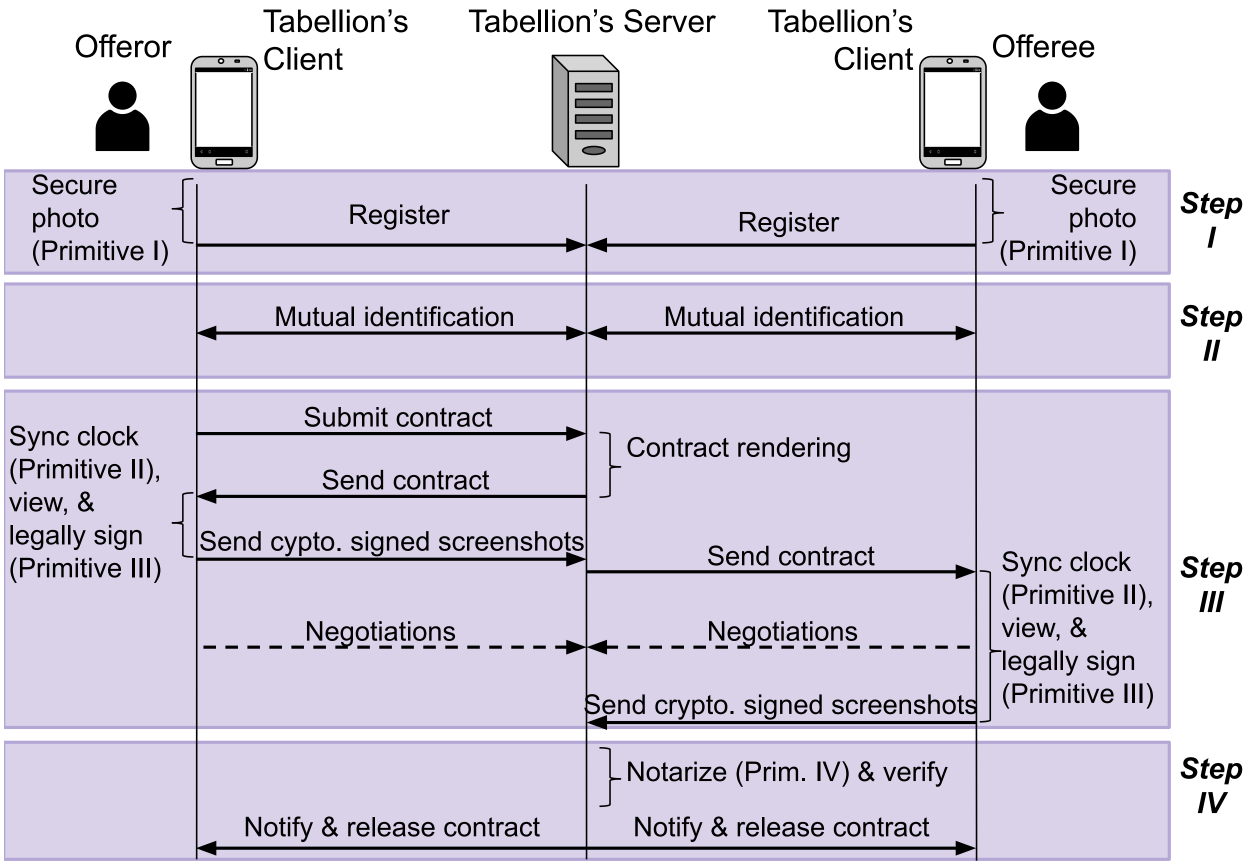

Offeror
Tabellion's Client
Tabellion's Server
Tabellion's Client
Offeree
Secure photo (Primitive I)
Register
Register
Secure photo (Primitive I)
Step I
Mutual identification
Mutual identification
Step II
Submit contract
Contract rendering
Sync clock (Primitive II), view, & legally sign (Primitive III)
Send contract
Send cypto. signed screenshots
Send contract
Sync clock (Primitive II), view, & legally sign (Primitive III)
Negotiations
Negotiations
Send crypto. signed screenshots
Step III
Notarize (Prim. IV) & verify
Notify & release contract
Notify & release contract
Step IV

Figure 6.2: Tabellion's contract formation protocol.

offeree whether they are expecting a contract from a named offeror (e.g., using an email address as the identifier). Once approved by the offeree, Tabellion uses the aforementioned securely-captured photos of the two parties to show to them. To ensure privacy, Tabellion does not exchange the photos before both parties approve having the intention of forming a contract with each other.

Step III: Legally signing a contract. In this step, the offeror first submits a contract to the server, which sends it to both parties to collect their legal signatures. Tabellion's server renders the contract into its own customized format, as described in §6.5.1, before sending it to the parties. The server also sends the certificate of the notary enclave to both parties so that they can include it in their signed screenshots (needed to prevent reuse of the screenshots). To legally sign the contract, Tabellion's mobile application first asks the TEE to use Primitive II to synchronize its clock (which will be needed to generate secure timestamps). It then uses Primitive III to display the contract to the user page by page and capture screenshots of content seen on the display. Note that this step may involve negotiations between the parties as discussed in §6.5.3. Finally, the application shows a special last page to each of the users that explicitly asks them to assent to (and hence sign) the contract (again using Primitive III).

Step IV: Notarizing the contract. Tabellion's server uses secure Primitive IV to cryptographically sign the contract as well as all the collected evidence. It then verifies the contract and releases it to both parties.

6.3.3 Self-Evident Contracts

Contracts in Tabellion are self-evident. That is, each user, and if needed, the court or an adjudicator, can independently verify the contract compliance with applicable law requirements.

A contract in Tabellion is formulated as

$$\{ \bigcup_{\mathtt{U}\in\mathtt{users}} (\{\mathtt{Photo_U}\}^{\mathtt{Pr_U}}, \{\mathtt{Pu_U}\}^{\mathtt{PrD_U}}, \mathtt{CertD_U}, \mathtt{MeasureD_U}, \bigcup_{\mathtt{i}\in\mathtt{pages}} (\{\mathtt{Screenshot_{i,U}}, \mathtt{ts_{i,U}}, \mathtt{Cert_N}\}^{\mathtt{Pr_U}})), \mathtt{Cert_N}, \mathtt{Measure_N}, \mathtt{ts_N}\}^{\mathtt{Pr_N}}$$

where $\{\mathtt{A}\}^{\mathtt{Pr}}$ indicates that $\mathtt{A}$ is cryptographically signed by private key $\mathtt{Pr}$, and $\bigcup_{\mathtt{U}\in\mathtt{users}} (...)$ and $\bigcup_{\mathtt{i}\in\mathtt{pages}} (...)$ represent union of users and contract pages, respectively.

The formula shows the components of a contract. $\{\mathtt{Photo_U}\}^{\mathtt{Pr_U}}$ is the photo captured using Primitive I, where $\mathtt{U}$ denotes either the offeror or the offeree. The photos are signed by the corresponding party's per-user per-device private key ($\mathtt{Pr_U}$). To verify this key, the contract also includes the corresponding public key ($\{\mathtt{Pu_U}\}^{\mathtt{PrD_U}}$), which itself is certified by a device-specific private key in the party's corresponding device TEE ($\texttt{PrD}_\texttt{U}$), and hence the contract also includes the certificates of the device TEEs of the two parties ($\texttt{CertD}_\texttt{U}$). The device TEE certificate is simply the public key of the device certified by the device vendor. More specifically, this is the device-specific ARM TrustZone certificate [58]. The contract also includes the measurements of the TEE code in the devices of the two parties ($\texttt{MeasureD}_\texttt{U}$). These measurements let the verifier know what software was running in the TEEs and are signed by the aforementioned device-specific keys.

The next components are the screenshots collected from the parties. The number of screenshots from each user can be different as the contract may be formatted differently for each user (§6.5.1). A signed screenshot also includes a timestamp captured with Primitive II ($\texttt{ts}_{\texttt{i},\texttt{U}}$), highlighting when the screenshot was confirmed by the user (as we discuss in §6.4.2, the timestamp comes with a confidence interval). It also includes the certificate of the notary ($\texttt{Cert}_\texttt{N}$). The latter is to ensure that each signed screenshot can be used for one contract only. Without it, an attacker may take a screenshot from a contract signed by a victim and try to include that in a different contract by the same victim.

The last couple of components are related to the notary. This includes the certificate of the notary, which is the certificate of the Intel SGX enclave [53]. It also includes the measurement of the code in the enclave code ($\texttt{Measure}_\texttt{N}$) and the time of notarization ($\texttt{ts}_\texttt{N}$). Finally, the contract is signed by the notary's private key ($\texttt{Pr}_\texttt{N}$).

6.3.4 Contract Verification Process

In Tabellion, one can verify the contract compliance with applicable law requirements as follows. To verify signature attribution, one needs to check that the same per-user per-device private key ($\texttt{Pr}_\texttt{U}$) is used to sign the photo ($\{\texttt{Photo}_\texttt{U}\}^{\texttt{Pr}_\texttt{U}}$) and the screenshots ($\{\texttt{Screenshot}_{\texttt{i},\texttt{U}}, ...\}^{\texttt{Pr}_\texttt{U}}$). Moreover, as described in §6.7.2, in Tabellion, we require a certain gesture to be performed in the photo to detect awareness. Therefore, one needs to check the presence of this gesture too.

To verify mutual assent, one needs to (*i*) check the content of the contract screenshots ($\texttt{Screenshot}_{\texttt{i},\texttt{U}}$) of the two users to make sure they both have the same content and (*ii*) check the timestamps of the screenshots ($\texttt{ts}_{\texttt{i},\texttt{U}}$), which include negotiations details (§6.5.3), to verify the order of the actions.

To verify that parties had an opportunity to read the contract, one needs to check that all contract pages are signed with the per-user per-device private key ($\{\texttt{Screenshot}_{\texttt{i},\texttt{U}}, ...\}^{\texttt{Pr}_\texttt{U}}$).

In addition, one needs to perform several more correctness checks. More specifically, one needs to check the public key of each user ($\texttt{Pu}_\texttt{U}$), to make sure all the cryptographic signatures by the clients are valid; check the certificate of the devices ($\texttt{CertD}_\texttt{U}$), to make sure that the users used a device verified by its vendor; check the certificate of the notary ($\texttt{Cert}_\texttt{N}$), to make sure a real enclave was used for notarization; check the software measurements of the device TEEs and the notary ($\texttt{MeasureD}_\texttt{U}$ and $\texttt{Measure}_\texttt{N}$), to make sure they used the expected code; check the inclusion of the notary certificate in the signed screenshots ($\{..., \texttt{Cert}_\texttt{N}\}^{\texttt{Pr}_\texttt{U}}$), to make sure the screenshots were not reused from another contract; check the notarization timestamp ($\texttt{ts}_\texttt{N}$), to make sure it is larger than the timestamps of all screenshots; and check the notary signature on the contract ($\texttt{Pr}_\texttt{N}$), to make sure the contract is correctly sealed.

6.4 Secure Realization of Primitives

Tabellion's self-evident contract assumes secure and untampered execution of the primitives. Therefore, it is critical to implement these primitives with a small amount of code so that their TCB remains small. A small TCB makes the primitives less prone to software bugs (which can get exploited by attackers). Moreover, a smaller code base can be easily inspected

Tabellion's Component	**Trusted Code**		**Untrusted Code**	
	Component	Size	Component	Size
Client	Primitive I	166	Mobile App	9919
	Primitive II	104		
	Primitive III	80		
	Shared	291		
Server	Primitive IV	185	Rest	4180
Combined		**826**		**14099**

Table 6.1: Tabellion's trusted and untrusted code size. The sizes are reported in LoC. We count the lines of the code we added, but not the existing code, e.g., TEE OS or Android libraries.

for safety and even certified. Table 6.1 shows that, while Tabellion needs a large amount of code (∼15,000 LoC) to implement all of its functionality, the size of trusted code is small (∼1,000 LoC). In this section, we discuss some of the important challenges we faced and solved to achieve this goal.

6.4.1 Primitive I: Secure Photo

Challenge. One straightforward way to implement this primitive is to exclusively control the camera in the TEE (a feature supported by ARM TrustZone). In this case, the TEE can directly take the photo and sign it. Unfortunately, this approach significantly bloats the TCB as it requires moving the camera device driver to the TEE. For example, in the Nexus 5X smartphone, the size of the camera driver is 65,000 LoC.

Solution. Our key idea to solve this problem is for the TEE to protect the camera photo buffer (rather than the whole camera driver) in memory from the time that the camera is about to capture the photo until when it is cryptographically signed. To protect the camera photo buffer, Tabellion write-protects the buffer pages before the camera device populates them with the photo data using Direct Memory Access (DMA). Moreover, to prevent the untrusted OS from storing a fake image in the camera photo buffer before

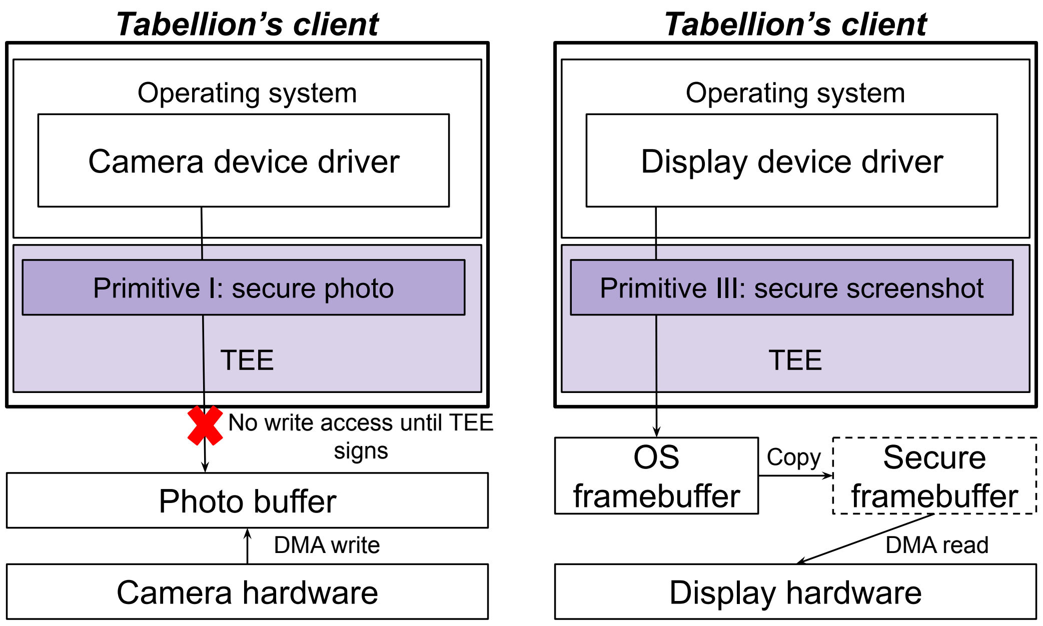


Figure 6.3: (Left) Secure realization of Primitive I. (Right) Secure realization of Primitive III.

protection, Tabellion zeroes out the buffer right after protection. Figure 6.3 (Left) illustrates this solution.

We implement two APIs in the TEE for this purpose. The application calls the normal OS photo capture API and the OS kernel uses the TEE API to capture a secure photo and returns it to the application. The kernel calls the first API, `prepare_photo_buffer`, to register a memory buffer to be used for secure photo capture. This API takes one argument, `photo_buf_paddr`, which is the physical address of the photo buffer in the OS physical address space. This API write-protects the buffer and zeroes out its contents. The kernel then waits for the camera hardware to populate this buffer through DMA. Next, the kernel calls the second API, `show_photo_buffer`, which displays the photo on the screen. Finally, the kernel calls the third API, `sign_photo_buffer`, which waits for the user to confirm the photo on the screen, and then cryptographically signs the photo and returns it using shared memory.

Note that to protect against the attacker using another DMA-enabled device to write to the photo buffer, Tabellion can use IOMMUs available in ARM SoCs, similar to Schrodin-Text [65].

6.4.2 Primitive II: Secure Timestamp

The TEE needs to be able to use a secure clock synchronized with a global clock. Network Time Protocol (NTP) is a popular protocol that can be used for clock synchronization. For security, we assume and use an integrity-protected channel (i.e., signed messages) to communicate with a secure NTP server, such as [45], to prevent a man-in-the-middle attack, which may tamper with the messages. This makes sure that the OS (or an attacker in the network) cannot change the content of the messages.

Challenge. Unfortunately, this security provision is not enough and an attacker can still mount an *asymmetric delay attack*. That is, the OS (or an attacker in the network) can delay the outgoing and incoming messages (from the TEE to the secure NTP server) in order to tamper with NTP calculations. We next describe this attack in more detail and then provide our solution.

In NTP, the client calculates its clock offset from the NTP server's clock as $\frac{(\mathtt{t}_2-\mathtt{t}_1)+(\mathtt{t}_3-\mathtt{t}_4)}{2}$, where $\mathtt{t}_1$ and $\mathtt{t}_4$ are timestamps captured by the client when it first sends a message to the NTP server and when it receives a response, and $\mathtt{t}_2$ and $\mathtt{t}_3$ are the timestamps captured by the NTP server when it first receives a message from the client and when it sends a response (which sends $\mathtt{t}_2$ and $\mathtt{t}_3$ to the client). This offset can then be used to synchronize the client clock. The NTP protocol assumes that the time to send a message from the client to the NTP server is the same as the time to send a message from the server to the client (hence the divide by two). An attacker can inject an *asymmetric delay* into one of these messages, e.g., using a compromised OS on the client or a compromised network link, in order to tamper with the calculated offset.

Solution. To address this challenge, Tabellion uses a novel secure clock synchronization strategy, built on top of NTP, which we call *delay-resistant NTP*. Our solution defeats the asymmetric delay attack by calculating a confidence interval, which represents the maximum and minimum possible offsets assuming arbitrary delay in any of the messages. Tabellion tags each action with its timestamp and confidence interval. For mutual assent, in addition to ordering the timestamps, Tabellion's contract verification requires that confidence intervals be non-overlapping.

To calculate the interval, we assume two extreme cases, one where only the request from the client to the NTP server forms the full round trip time (and the response takes no time) and one vice versa. It is possible to show that $\mathtt{offset_{max}} = \mathtt{max}(\mathtt{t_3} - \mathtt{t_4}, \mathtt{t_2} - \mathtt{t_1})$ and $\mathtt{offset_{min}} = \mathtt{min}\ (\mathtt{t_3} - \mathtt{t_4}, \mathtt{t_2} - \mathtt{t_1})$. Therefore, the confidence interval (ci) is calculated as $\mathtt{ci} = |(\mathtt{t_3} - \mathtt{t_4}) - (\mathtt{t_2} - \mathtt{t_1})|$.

We add two APIs to the TEE for this primitive. They allow the application to initiate the protocol and to communicate with the NTP server. The application calls the first API, `sync_clock_init`, to initiate the protocol. This API returns a nonce from the TEE (used to protect against replay attacks). The application then forwards the nonce to the NTP server and forwards the response from the NTP server to the TEE with a call to the second API, `sync_clock_complete`. This API takes one argument, `server_ts`, which is a shared buffer for passing the two server timestamps in NTP protocol and the server's signature (RSA with a 1024 bit key).

Note that in addition to a secure synchronization mechanism, the TEE needs a secure hardware timer to keep track of time after synchronization. For that, we use a secure hardware timer available in TrustZone.

6.4.3 Primitive III: Secure Screenshot

Challenge. The TEE needs to securely capture the content on the display. A straightforward approach to achieve this is to give exclusive control of the display subsystem to the TEE. Unfortunately, doing so requires moving the display subsystem driver to the TEE, which on the HiKey board, encompasses at least 8,000 LoC. This bloats the TCB.

Solution. Our key idea in Tabellion is to secure the buffer used for displaying content (i.e., framebuffer) in the TEE rather than the whole display software stack. At a high-level, the primitive is realized as follows. When invoked, the TEE *freezes* the framebuffer, not allowing any more updates. It then waits for the user's authentication using biometrics. Once the user confirms, the TEE signs a copy of the framebuffer and *unfreezes* it. This process guarantees that the displayed content and the authentication are atomic.

Disallowing updates to the framebuffer can break the display stack in the OS. Therefore, the TEE copies the contents of the framebuffer to a newly generated framebuffer (i.e, secure framebuffer), which is only accessible in the TEE. It then shows the secure framebuffer on the display by programming its address into the memory-mapped display controller register that holds the address of the framebuffer. Furthermore, to prevent a compromised OS from overwriting this register (in order to use a different framebuffer), Tabellion also removes write permission from the corresponding register page of the display controller. With this solution, the OS is allowed to update its own framebuffer but doing so does not change the content shown on the display. Upon unfreezing the display, Tabellion points the display controller back to the untrusted framebuffer and enables writes to the aforementioned display controller register. Figure 6.3 (Right) illustrates this approach.

Note that when the TEE removes the permission of the above register page, any write to this register page of the display controller would fault. Indeed, there are other registers on the page as well, all of which will be write-protected. To avoid these faults, we made minimal changes to the display controller driver in the OS to skip the writes while the display is frozen.

To provide this primitive, the TEE exposes three APIs. It expects the application (through the OS) to call these APIs. First, to show a contract page, the application displays the page and then makes a call to TEE's *freeze_framebuffer*, which freezes the framebuffer showing the contract page. The TEE waits for the user's confirmation using biometrics. Note that in commodity mobile devices, biometric devices are controlled in the TEE [47], therefore, we assume so in our design. Once the user confirms, the application makes a call to the second API, *sign_framebuffer*, which captures a screenshot in the TEE and cryptographically signs it, appending the secure timestamps and the notary certificate passed with the API. This API takes the notary certificate as input and returns the signed framebuffer using shared memory. Finally, the application makes a call to the third API, *unfreeze_framebuffer*, which unfreezes the framebuffer.

As one last provision, we require each page shown through this primitive to stay on the display for a minimum of 2 seconds. This prevents a user from confirming a page by mistake without having enough time to read it.

6.4.4 Primitive IV: Secure Notarization

Challenge. Collecting one piece of evidence poses a challenge for the notary. Specifically, at the time of notarizing the contract, the notary enclave does not know for certain whether the offeror has revoked the offer or not. The law of contracts recognizes the offeror's right to revoke the offer as long as it is not signed by the offeree and this revocation is otherwise permitted under the terms of the offeror's offer. For example, the offeror might have revoked the offer 5 minutes prior to the offeree legally signing the contract (hence not satisfying the requirement of mutual assent), but the revocation evidence might not reach the notary in time.

Solution. To address this problem, the enclave requires a confirmation from the offeror that there have been no revocations, and if there has been one, it requires the secure screenshot confirming that. The enclave, through the rest of the server code, inquires about any pending revocations in the offeror and waits until it receives the response.

Note that this solution, while secure, might cause a practical problem. That is, no response from the offeror's device can stall the notarization of the contract indefinitely. To prevent indefinite blocking, a possible approach is to wait for no longer than a configurable period of time, e.g., 24 hours. This allows the offeror's device to send the response in most practical cases.

6.5 Fully Functional Platform

The secure primitives that we design and build for Tabellion, while effective in providing strong evidence for the contract, pose challenges for building a fully functional contract platform. We discuss all the challenges we faced and solved by developing ∼14,000 untrusted lines of code in our client and server.

6.5.1 Readable Contracts

Challenge. Contracts are often in PDF or Word formats, with each page containing a large amount of text. When viewed on the display of a mobile device, users need to continuously *pinch and zoom* to fully view the content. Doing so creates challenges for the use of Primitive III (secure screenshot) to capture all the content seen by the user.

Solution. We solve this problem by providing a service in Tabellion's server that renders the contract into a readable one when viewed fullscreen on the mobile device of each user. This service accepts a contract in an intermediate format (Markdown language in our prototype). It generates the contract pages specifically for the screen sizes of the mobile devices used by the offeror and the offeree. Moreover, the service follows some formatting guidelines for the rendered document to make sure the content is easily readable. These guidelines include adequate line spacing, margins, clear background color, and font colors. Indeed, it is possible to add a feature to the service so that it can apply user-specific requests, e.g., a font size larger than the default one. In addition, the server clearly marks every page with a page number, which helps verify the presence of all required screenshots in the contract.

We note that contract rendering in the server is not part of the TCB of the system. This is because Tabellion asks the user to read through and sign the generated contract.

6.5.2 Contract Submission

Challenge. As mentioned, Tabellion's contract generator receives the contract in an intermediate format, such as Markdown. This creates a burden for the offeror, who may prefer another format to prepare the contract in.

Solution. To address this problem, we provide a mostly-automatic converter. More specifically, we allow the offeror to submit the contract in PDF format. Our converter then extracts the content from the PDF file and converts it to a mobile app UI page with explicit headers and text sections. The offeror is then allowed to review the extracted content and edit it, if needed. Once finalized, the converter produces a Markdown file and sends it to the server. Figure 6.4 shows this solution with an example.

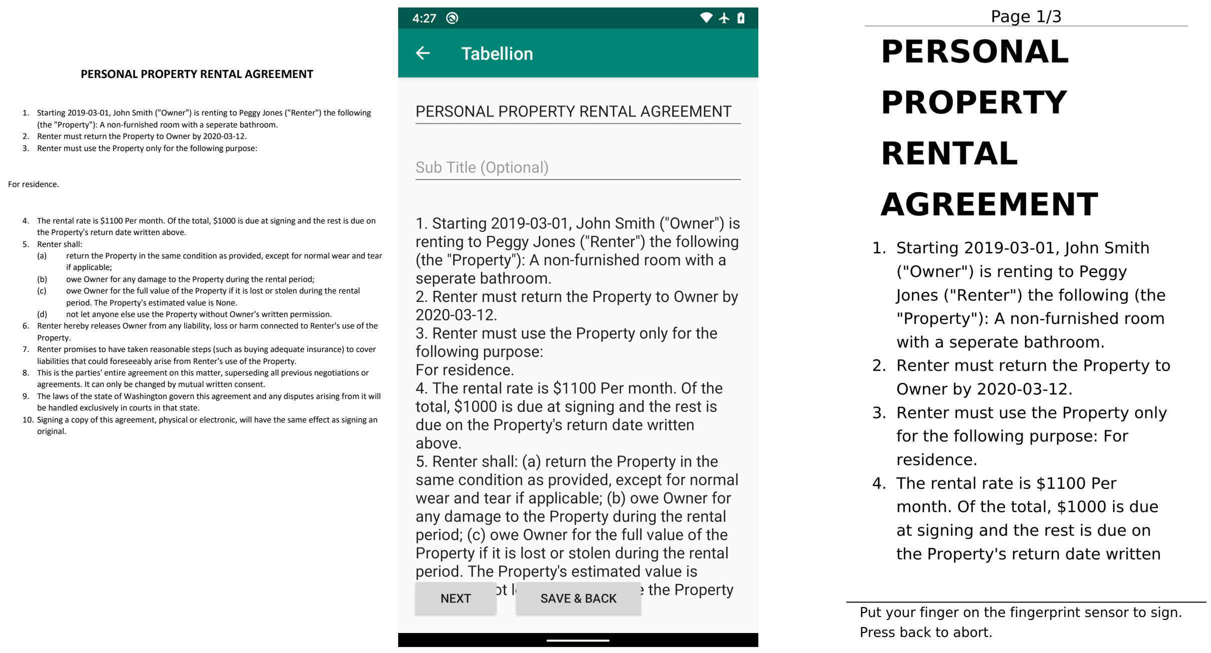
We do, however, note that our prototype cannot currently handle complex PDF pages (i.e., those with images and tables). We leave addressing this limitation to our future work.

6.5.3 Contract Negotiations

Challenge. When parties negotiate and vary the terms of a contract, the parties will need to mutually assent to the new terms if they are materially different from the terms of the original contract. In order to demonstrate mutual assent, one needs to provide evidence of these negotiations. Unfortunately, creating an out-of-bound channel to allow the parties to perform negotiations, e.g., messaging apps or email, require other primitives to securely capture the negotiations.

Solution. We have implemented a fully-functional negotiation interface in Tabellion using the existing primitives. To achieve this, our Tabellion application allows the offeree to enter a comment on the offer. The application then shows the comment on a new UI page to the offeree and asks them to confirm the comment, similar to how they confirm seeing a contract page. Tabellion then uses Primitive III to capture a confirmed screenshot of the revision request and includes it in the contract. Finally, this revision request is shown to the offeror, who can revise the contract and submit again, using the Android UI page described earlier and seen in Figure 6.4 (Middle). Note that the details of negotiations can be easily verified in a Tabellion contract by inspecting the confirmed screenshots of the contract pages and the revision requests along with their timestamps.

Opportunity. The combination of the previous three solutions has enabled us to add an important capability to Tabellion that no existing platform supports: *negotiation integrity tracking*. More specifically, after the offeror edits the original offer and submits it, Tabellion's server compares the edited contract with the old version and identifies the contract pages that are affected by changes. It then asks the offeree to only view and modify these edited



PERSONAL PROPERTY RENTAL AGREEMENT

1. Starting 2019-03-01, John Smith ("Owner") is renting to Peggy Jones ("Renter") the following (the "Property"): A non-furnished room with a seperate bathroom.
2. Renter must return the Property to Owner by 2020-03-12.
3. Renter must use the Property only for the following purpose:

For residence.

4. The rental rate is $1100 Per month. Of the total, $1000 is due at signing and the rest is due on the Property's return date written above.
5. Renter shall:
   (a) return the Property in the same condition as provided, except for normal wear and tear if applicable;
   (b) owe Owner for any damage to the Property during the rental period;
   (c) owe Owner for the full value of the Property if it is lost or stolen during the rental period. The Property's estimated value is None.
   (d) not let anyone else use the Property without Owner's written permission.
6. Renter hereby releases Owner from any liability, loss or harm connected to Renter's use of the Property.
7. Renter promises to have taken reasonable steps (such as buying adequate insurance) to cover liabilities that could foreseeably arise from Renter's use of the Property.
8. This is the parties' entire agreement on this matter, superseding all previous negotiations or agreements. It can only be changed by mutual written consent.
9. The laws of the state of Washington govern this agreement and any disputes arising from it will be handled exclusively in courts in that state.
10. Signing a copy of this agreement, physical or electronic, will have the same effect as signing an original.

PERSONAL PROPERTY RENTAL AGREEMENT

Sub Title (Optional)

1. Starting 2019-03-01, John Smith ("Owner") is renting to Peggy Jones ("Renter") the following (the "Property"): A non-furnished room with a seperate bathroom.
2. Renter must return the Property to Owner by 2020-03-12.
3. Renter must use the Property only for the following purpose:
For residence.
4. The rental rate is $1100 Per month. Of the total, $1000 is due at signing and the rest is due on the Property's return date written above.
5. Renter shall: (a) return the Property in the same condition as provided, except for normal wear and tear if applicable; (b) owe Owner for any damage to the Property during the rental period; (c) owe Owner for the full value of the Property if it is lost or stolen during the rental period. The Property's estimated value is
ot l the Property

Page 1/3

PERSONAL PROPERTY RENTAL AGREEMENT

1. Starting 2019-03-01, John Smith ("Owner") is renting to Peggy Jones ("Renter") the following (the "Property"): A non-furnished room with a seperate bathroom.
2. Renter must return the Property to Owner by 2020-03-12.
3. Renter must use the Property only for the following purpose: For residence.
4. The rental rate is $1100 Per month. Of the total, $1000 is due at signing and the rest is due on the Property's return date written

Put your finger on the fingerprint sensor to sign.
Press back to abort.

Figure 6.4: (Left) A contract sample in PDF. (Middle) Extracted contract in Tabellion presented in an Android UI activity, which allows edits. (Right) Contract rendered by Tabellion and viewed on the device. The original contract is a single page with small fonts, which is barely readable on a smartphone screen in fullscreen mode. The converted contract has 3 pages and is easily readable.

pages. This capability provides important usability benefits, especially when dealing with long contracts. In today's platforms, this is left to the offeree. That is, the offeree can decide to view the parts of the revised contract that they think have been updated. However, they bear the risk of not seeing other changes added (possibly maliciously) to the contract. Alternatively, they can re-read the whole revised contract again, which is time-consuming, especially if there are multiple rounds of negotiations.

6.5.4 Automatic Contract Verification

Tabellion's contracts are self-evident. Yet, the verification process is not easy and requires several checks. Therefore, to enhance the usability of Tabellion, we provide an automatic contract verifier. We use this verifier in our own server to verify the contract once it is formed and before notifying the users. We note that contract verification in the server is not part of the TCB of the system. This is because each user can independently verify the contract as well.

Challenge. The rendering of the contract specifically for each user creates a challenge for automatic verification. That is because the contract pages (but not the content) might be different for each user (e.g., different number of pages, different page dimensions, and different font sizes). To check that both parties assent to the same contract terms, we cannot trivially compare the two sets of screenshots pixel by pixel.

Solution. To enable automatic verification, Tabellion's server releases some metadata alongside the notarized contract. This metadata includes information about the code used to render to the contract from the intermediate language (e.g., Markdown), the contract source in that intermediate language, and the format used for each mobile device (i.e., screens size, font size, etc.). The verifier uses the same generator code to render the contract pages from the source to the final pages for each mobile devices (which are PNG images as described in §6.6). It then compares these rendered images, pixel by pixel, with the contract screenshots signed by the users' devices. If the images fully match, verification is successful.

6.6 Implementation

Tabellion's client. We build Tabellion's client on a HiKey LeMaker development board. The TEE in this board is the Xen hypervisor (version 4.7) and the OPTEE OS (version 3.3) running in ARM's TrustZone secure world. We implement Primitives I and III (other than the cryptographic signatures) in the Xen hypervisor. We implement cryptographic signing operations as well as Primitive II in OPTEE. We use RSA with 2048 bit keys for digital signatures in the client.

We note that virtualization hardware extension is available in most of the ARM mobile SoCs that are used in current mobile platforms and, hence, adding a hypervisor is feasible. Indeed, some mobile manufacturers have already added a hypervisor layer for security purposes. For example, Samsung uses a hypervisor for real-time kernel protection as part of Samsung Knox [58].

We use a USB camera and a USB fingerprint scanner with the board and program them in the normal world. We use Android Open Source Project (AOSP) Nougat for the untrusted OS. The TCB size in this prototype is the trusted code that we added (Table 6.1) and the existing trusted code in TrustZone secure world and hypervisor (which can be as low as a few tens of thousands of lines [148]).

We also provide a secondary prototype of Tabellion for commodity mobile devices. We use this prototype for our user study and for energy measurements (§6.8). The main difference is the implementation of secure primitives. On commodity mobile devices, we cannot program the TEE, therefore, we emulate these primitives in the mobile app itself. In this prototype, we use the smartphone's camera and fingerprint scanner.

Tabellion's server. We process the contract in Markdown format and generate the contract pages as images in PNG format. We also attach instructions and page numbers to contract page (Figure 6.4 (Right)).

We implement the notary enclave in an Azure Confidential Compute Standard DC4s VM. This VM runs on top of the 3.7GHz Intel XEON E-2176G processor, which supports Intel SGX. We program the enclave using the open source Confidential Consortium Framework (CCF). For the measurement of the TCB and the enclave certificate, we use the Intel SGX Data Center Attestation Primitives (DCAP) libraries, which leverage Elliptic Curve Digital Signature Algorithm (ECDSA). We use RSA with 4096 bit keys for digital signatures by the notary in the enclave.

6.7 Security Evaluation

6.7.1 Threat Model

We assume that the Tabellion client's TEE is uncompromised. We assume that the attacker can access the victim's device (e.g., by stealing it) but cannot compromise its TEE. We do not trust Tabellion's application running in the user's device. We assume that the enclave in Tabellion's server is uncompromised. However, we do not trust the rest of the server components. We also assume that the attacker cannot leverage side channels to perform side-channel attacks on our TCB in the client TEE [131] and SGX enclave [87, 171].

We assume a secure NTP server, such as [45], with which clients can synchronize their clocks (§6.4.2). We trust the hardware of mobile devices, e.g., the camera, and the SGX feature of processors in the server. A self-evident contract includes certificates from the mobile device vendor (e.g., Samsung) and the enclave vendor (e.g., Intel). We trust these vendors.

Tabellion's prototype does not currently provide availability or confidentiality guarantees. Lack of the availability guarantee means that Tabellion's services, e.g., the registration service, may not to be available to users or that a contract signed with Tabellion may be lost. Lack of the confidentiality guarantee means that an attacker can access the content of a contract. These guarantees can be provided using existing solutions, e.g., encryption.

6.7.2 Security Analysis

We next analyze various attacks introduced in §6.2 and discuss whether they would succeed or fail against Tabellion.

Repudiation attack. We introduced three forms of this attack. In the first form, the attacker denies the signature. This would fail against Tabellion as the contract provides the photo of the user, signed with a key, which is authenticated with the user's biometrics and which also is used to confirm the contract pages. Moreover, the validity of the key can be verified by inspecting its certificate and the certificate of the mobile device. In the second form, the attacker denies mutual assent. This would fail as the screenshots confirmed by the user clearly show the contract and negotiation terms. Moreover, all screenshots are securely timestamped, which provides strong evidence of the order of actions. In the third form, the attacker denies that there was an opportunity to read the contract. This would fail as the contract includes signed screenshots of all the content viewed and confirmed by the user.

In addition, in either of these attack forms, the attacker may deny the strong evidence by Tabellion and claim their device or Tabellion's server was compromised. Tabellion's small TCB in its client and server provides strong protection against such claims. Moreover, the self-evident contract provides strong evidence that the expected code executed in the device TEE and enclave by providing their code measurements and certificates.

Impersonation attack. We introduced two forms of this attack. In the first form, the attacker must spoof the victim's authentication on the victim's device. This is challenging in Tabellion as it requires defeating TEE-protected biometric sensors with sophisticated anti-spoofing (e.g., Apple's Touch ID [63]).

An attacker in close proximity to the victim may attempt to defeat a fingerprint-based authentication by pressing the victim's fingers against the fingerprint scanner. While this is a difficult attack already, we note that it is feasible only if fingerprint is used as the sole biometric signal. Tabellion's design is conducive to using different or multiple biometric signals, e.g., Apple's Face ID [63]. Note that these modern biometric sensors are accessible in the mobile device TEE [58, 54] and hence using them does not require adding more trusted code.

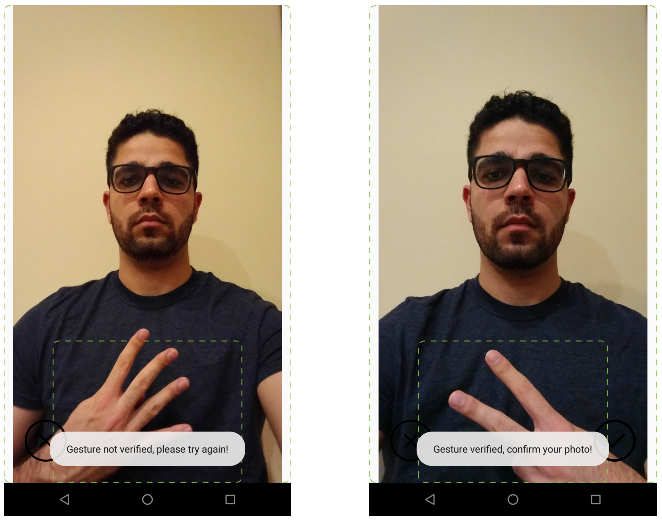

Gesture not verified, please try again!
Gesture verified, confirm your photo!

Figure 6.5: Custom gesture (V sign) required in Tabellion's photos. (Left) The user performs an incorrect gesture. (Right) The user performs the correct gesture. In both cases, the user is notified accordingly. In (Left), the notification says "Gesture not verified, please try again!" In (Right), it says "Gesture verified, confirm your photo!"

For the second form (where the attacker tries to spoof the victim's identity), we see five attack variants on Tabellion. We first briefly introduce these variants and then describe how Tabellion protects against them. The first variant is using an *existing or deep-faked photo of the victim*. The second one is taking *a photo of an unaware victim*. The third one is taking *a photo of a 3D-printed object looking like the victim*. The fourth one is taking *a photo of an existing or deep-faked photo of the victim shown on a display*. The fifth one is taking *a photo of a doppelganger*.

Tabellion defeats the first attack variant by its use of a secure photo, which guarantees that the photo is captured by the camera hardware. The second variant is challenging for the attacker as it requires physical proximity to the victim. Yet, Tabellion further defeats this attack by mandating a requirement for the photos used for registration: *specialized photo*. More specifically, Tabellion requires the user to perform a custom gesture while taking the secure photo in order to demonstrate *awareness*. The contract is not valid if the requirement is not satisfied and Tabellion's server automatically checks the requirement using an open source framework [50]. Figure 6.5 shows this solution in practice.

The third and fourth variants are also difficult for the attacker. Yet, Tabellion can make these attacks even harder by detecting *liveness*, using one of the several existing solutions [63, 43, 129, 170, 130]. For example, it can require the user to take multiple photos from different angles of their face, which can be used to detect liveness [43]. Alternatively, it can use an iris scanner or secure face recognition on modern phones, e.g., Apple's Face ID, which projects a grid of 30,000 infrared dots on a user's face and takes an infrared picture [63].

The fifth variant is also challenging for the attacker since it requires a doppelganger. Yet, Tabellion can defend against this attack by asking the user to show an official ID in the photo, which can be automatically checked using services such as Checkr [44].

Confusion attack. In this attack, a malicious offeror leverages vulnerabilities in the contract viewer in the offeree's device to mislead the offeree. Tabellion's rendering of the contract in its server neutralizes the Dalì attack (which is a specialized form of the confusion attack, discussed in §6.2) since the attacker cannot directly send a file to the offeree's device. Moreover, Tabellion's use of TEE to securely display contract pages defeats a powerful attacker who may be able to take control of the contract viewer on the victim's device (e.g., by compromising Tabellion's app or the OS) and try to change the contract content shown to and confirmed by the victim.

6.7.3 Case Analysis

In §6.1, we discussed a few legal cases where existing electronic contract platforms have failed to provide strong evidence. While it is difficult, if not impossible, to predict the legal validity of a contract with certainty, we believe Tabellion, if used for forming those contracts, would have provided stronger evidence. First, in *In re Mayfield* [33], the court discussed the ease with which the signature on DocuSign could be forged. It mentioned that "*what happens when a debtor denies signing a document and claims his spouse, child, or roommate had*

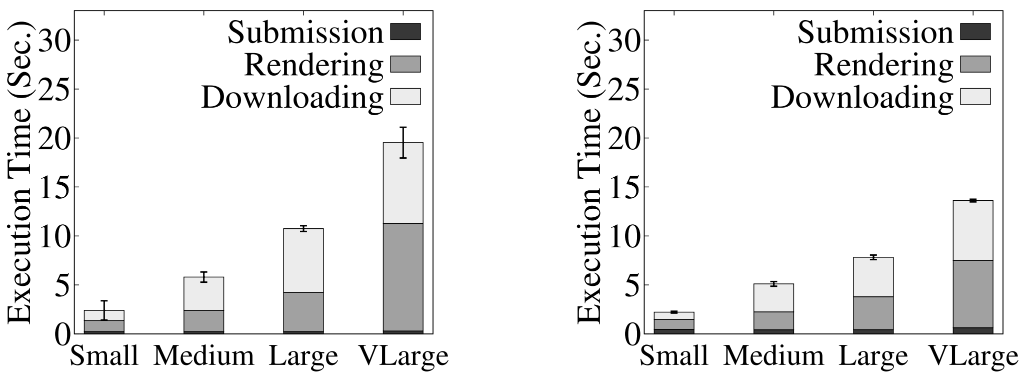


Figure 6.6: Tabellion's execution time of offer (as defined) for (Left) HiKey and (Right) Nexus 5X.

access to his computer and could have clicked on the 'Sign Here' button." In contrast, Tabellion provides a secure photo of the signatory and protects against impersonation attempts, as discussed.

Second, in *Adams v. Quicksilver, Inc.* [41, 28], the problem was that "*it couldn't be determined when the agreement was signed.*" Tabellion's use of secure timestamps for every action in the signing process provides strong evidence for when each party signed the contract.

Finally, in *Labajo v. Best Buy Stores* [140, 27], the problem was that "*Best Buy couldn't prove that she [Christina] saw and approved the disclosure.*" Tabellion's use of secure screenshots to capture all the content seen by a user provides strong evidence for this requirement.

6.8 Evaluation

6.8.1 Performance Evaluation

We present the execution time of using Tabellion (measured on the client device). We include results for our main prototype on the HiKey board and our prototype on a Nexus 5X smartphone.

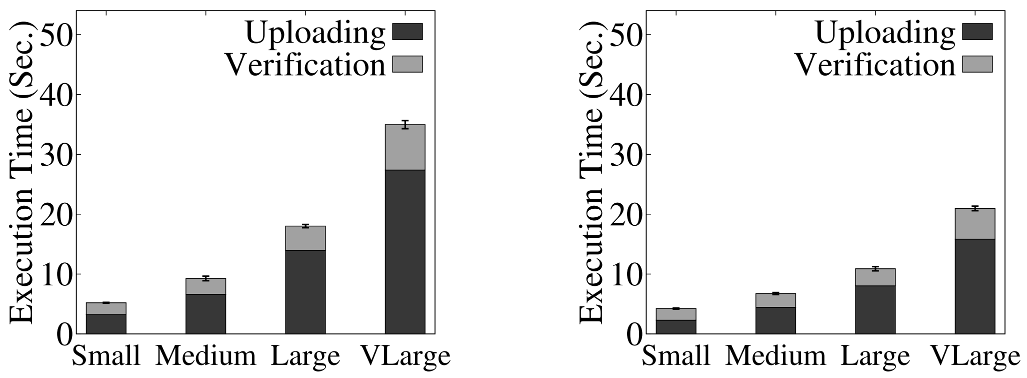


Figure 6.7: Tabellion's execution time of acceptance (as defined) for (Left) HiKey and (Right) Nexus 5X.

Figure 6.6 shows the *execution time of offer*, defined and measured from when the offeror submits a contract to Tabellion until when the contract is ready for them to sign. This includes the time needed to send a request to the server, render the contract pages in the server, and download them to the device. The figure shows the results for four contracts of different lengths. These contracts, labeled as small, medium, large, and very large, result in 6, 12, 24, and 48 pages for HiKey and 4, 8, 16, and 32 pages for Nexus 5X (the numbers are different in the two platforms since they have different screen sizes). As can be seen, even for very large contracts, the overall time is less than 20 seconds.

The same figure also shows the breakdown of the execution time. It shows it is mostly due to transfer of contract pages from the server to the device and due to rendering of the contract. Our rendering pipeline can be improved, as it currently renders the contract in several stages (Markdown to HTML, HTML to PDF, and finally PDF to PNG).

Figure 6.7 shows the *execution time of acceptance*, defined and measured from the time that the offeree submits their signed contract to Tabellion until when the contract is notarized and verified (excluding the last inquiry to the offeror, which might take very little time or an arbitrarily long time depending on the reachability of the offeror, as discussed in §6.4.4). The results show that the execution time is around 35 seconds for the very large contract.

As the results show, most of the execution time is due to uploading the signed screenshots to the server.

We next measure the execution time of secure primitives on the HiKey board (Primitives I-III) and on the server (Primitive IV). More specifically, we measure the execution time of a single use of a secure primitive. For each primitive, we measure the execution time several times and report the average and standard deviation. Figure 6.8 shows the results. It shows the execution times of these primitives are small. Note that Primitive III enforces at least a two second display freeze (§6.4.3).

6.8.2 Energy Measurement

We measure the energy consumption of Tabellion in our secondary prototype with a Nexus 6P smartphone. We perform this experiment just to demonstrate that Tabellion does not drain the battery, which would cause inconvenience to the user. We measure the energy on Nexus 6P as opposed to Nexus 5X, which we use in other experiments. This is because the fuel gauge in Nexus 6P, unlike Nexus 5X, includes a charge counter [55]. We measure the energy for the offeror submitting and signing a very large contract (28 images on Nexus 6P). Our results show that the overall energy consumption is on average 68.97 (standard deviation of 4.67) milliwatt-hours, which is 0.5% of the overall battery capacity on this smartphone assuming a 3.7 V voltage.

6.8.3 User Study

We perform an IRB-approved user study[5] to evaluate four usability aspects of Tabellion: effectiveness of presentation, usage convenience, readability of contracts, and time spent

[5]UC Irvine IRB HS# 2019-5017

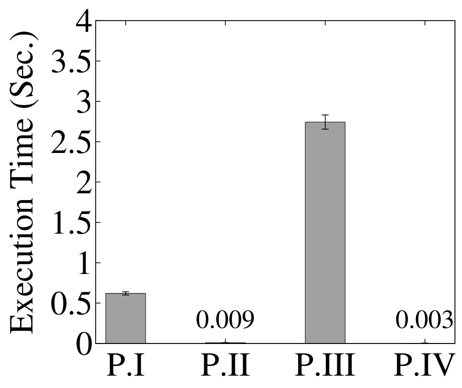


Figure 6.8: Execution time of secure primitives.

on contracts. Note that we do not evaluate other usability aspects of the system, such as registration.

We recruit 30 participants in our study (22 undergraduate and 8 graduate computer science students; 21 male and 9 female). We ask the participants to read and sign contracts on a Nexus 5X smartphone. After each contract, we ask them to answer two multiple-choice questions about the contract's details and ask them to rate the convenience and readability of the contract platform. Example questions are true/false questions about a specific statement in the contract or a question asking about a specific value, e.g., the total cost of a service. Each participant uses two different platforms, Tabellion and DocuSign [48], and signs four contracts on each. We choose the contracts out of a repository of eleven contracts and rotate the contracts among platforms and participants to avoid any systemic bias (i.e., each contract is signed by several users on each platform). These contracts include sale, loan, Non-Disclosure-Agreement (NDA), and rental contracts that we created using contract samples from Docsketch [46].

Effectiveness of presentation. To measure the effectiveness of presentation and user understanding, we count correct and wrong answers to the questions for the contracts. Figure 6.9 (Left) shows the results. It shows that the participants fared slightly better in Tabellion, demonstrating that Tabellion's rendering of the contract specifically for a mobile device and the use of good formatting guidelines (§6.5.1) are effective.

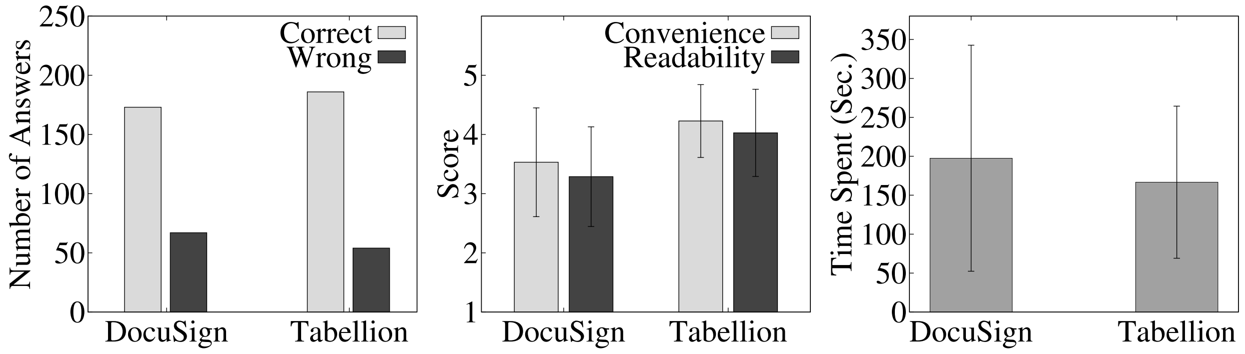


Figure 6.9: (Left) Correct/wrong answers for questions in the user study. (Middle) Convenience and readability of platforms rated by participants. (Right) Average time spent on the contracts in the user study.

Convenience and readability. Using a Likert Scale, we ask the participants to rate the convenience and readability of the platforms. Figure 6.9 (Middle) shows that the participants slightly preferred the convenience and readability of Tabellion over DocuSign for signing the contracts. More specifically, on average, the participants scored the convenience and readability of the contracts to be 4.22 and 4.02 for Tabellion and 3.53 and 3.28 for DocuSign.

Time spent on contracts. Figure 6.9 (Right) shows that participants spent slightly less time to view and sign the contracts on Tabellion(166.68 seconds on Tabellion and 197.46 on DocuSign, on average).

All of these results show that the security benefits of Tabellion do not come at the cost of usability. However, one might wonder whether Tabellion's usability decreases for long contracts. We report the results for the longest contract in our study, which was an NDA contract, consisting of 7 PDF pages for DocuSign and 25 pages for Tabellion. Our results show that participants scored the convenience and readability of the NDA contract, respectively, to be 4 and 3.6 for Tabellion and 2.6 and 3.1 for DocuSign. Moreover, they spent 355 seconds for this contract on Tabellion and 490 seconds on DocuSign. This shows that the usability of Tabellion is good even for long contracts.

Chapter 7

Related Work

7.1 TEE-based Security Systems

Several systems use virtualization and TrustZone for security purposes. DelegaTEE [139] uses TrustZone and SGX to provide delegation of credentials. TruZ-Droid [178] enhances the functionality of TrustZone TEE by providing a binding between an Android app and a Trusted Application (TA) so that the Android app callback functions can be triggered from a TA. TrustFA [182] designed a remote facial authentication method that, unlike Tabellion, moves the camera and display driver into the TEE, which increases the TCB size. Trusted sensors [132, 110] attest sensor data to applications. Gilbert et al. [110, 111] attest the integrity of sensor data. fTPM [159] implements the TPM functionality within TrustZone. Samsung Pay [59] authenticates users via fingerprints in TEE for confirming a transaction. Android Protected Confirmation [35] provides APIs for applications, such as a banking app, to get the user's confirmation on certain important messages, such as those used for transferring money. TrustDump [169] uses ARM TrustZone to record the memory and register space of the normal world operating system when it crashes or is compromised. TrustZone-based

Real-time Kernel Protection (TZ-RKP) [75] used in Samsung KNOX [29] uses ARM TrustZone to implement memory protection solutions for smartphones. TrustZone has also been used for secure credentials [123], secure facial authentication [182], and safe execution of applications (by offloading part of the application to the secure world) [163]. SPROBES [108] intercepts certain events in the normal world operating system to trap in the secure world in order to perform introspection of the operating system. This is performed by rewriting certain instructions in the operating system image to make an SMC call to monitor the normal world, however, we do so by trapping accesses to hypervisor, requiring no operating system modifications. Several works also use SGX enclave to provide different security guarantees [118, 74]. VButton [128] provides a framework for attesting the user operations in different apps. TruZ-View [179] provides UI integrity and confidentiality protection without porting the UI renderer to the TEE. AdAttester [127] provides attestation for the user's click on ads in Android apps. SchrodinText [65] provides text view confidentiality protection using hypervisor and TrustZone TEE. Yu et al. [181] and Zhou et al. [184] designed trusted path for GPUs to protect display authenticity and confidentiality. None of these systems use virtualization hardware and TrustZone to provide security services similar to Viola, Ditio, or Tabellion.

7.2 Virtual Machine Introspection

Virtual machine introspection (VMI) uses the hypervisor to monitor the operating system for intrusion detection [107, 105, 100]. It provides good visibility of the operating system internals while protecting the monitoring system from the attacks on the operating system. Similar to VMI, in Viola, we leverage the hypervisor to enforce the I/O invariants eliminating the need to trust the operating system. Our work in Viola is, however, fundamentally different as, unlike VMI, we provide guarantees on the correct behavior of I/O devices.

7.3 Untrusted Operating System and Drivers

A compromised operating system can attack the user space applications. This observation has fueled research into protecting the applications from a compromised operating system. Most solutions, such as Overshadow [90] and InkTag [116], do so with a trusted hypervisor, which mediates applications' interactions with the operating system. However, due to the wide and complex interface between an application and the operating system, these solutions cannot protect the applications against all possible attacks, such as Iago attacks [85]. Similarly, we leverage the hypervisor in the security monitor in order to make the operating system untrusted.

Nexus [175] makes the device drivers untrusted by running them in user space and by vetting their interactions with the I/O devices. Similar to Viola, Nexus adopts a domain-specific language to write specifications for the devices. There are key differences between Nexus and Viola. To mention a few, Unlike Nexus, Viola's compiler is formally verified. Moreover, Nexus is implemented for the x86 architecture and PCI devices, Viola is implemented for the ARM architecture and non-PCI devices. Despite these differences, Viola can benefit from the design of Nexus by moving the device drivers to user space. Such a design will enhance the security guarantees of Viola's kernel-based monitor as it removes the device drivers out of the TCB.

In this line of work, several other work use Intel SGX or ARM TrustZone to protect the user space applications from the untrusted operating system. Haven [78] executes Windows applications in the enclave by using Library OS. Ryoan [118] sandboxes the untrusted code in the enclave to protect user's secret in the enclave. SCONE [74] provides user enclave from the containers to protect the the user space containers from the untrusted operating system. SecTEE [183] and SANCTUARY [81] use TrustZone to provide enclave for applications running in ARM SoC-based devices. While these line of work focus on protecting user space applications from the untrusted operating system using SGX enclaves, security monitor focuses on protecting certain security services.

7.4 Other Related Work

7.4.1 Performance of Virtualization

Some existing work evaluates the performance of virtualization and provides novel designs. Dall et al. [96] evaluates the performance of ARM virtualization with a focus on ARM servers. Cherkasova et al. [91] measure the performance of I/O activities in Xen. Gehrmann et al. [109, 101] similarly compare the use of virtualization and ARM TrustZone on mobile phones, but do not provide performance measurement on a real hardware. OKL4 microvisor [114] is a hypervisor that has the flexibility of a microkernel. It is built based on multiple isolated components and supports multiple virtual machines. NOVA [168] also has a similar idea but for x86 systems and minimizes the code base by moving the virtualization support code to the user level. Cells [66] virtualizes Android at the operating system level to provide multiple Android phones with seperate phone numbers without using ARM virtualization hardware. LightVM [138] modifies Xen and its tools to have performance comparable to containers for certain operations such as virtual machine boot time and migration. However, we focus on improving the performance of the system when virtualization is used to provide a TEE.

7.4.2 Software Verification

An alternative approach to Viola for implementing trustworthy sensor notifications is to guarantee that there are no bugs in the whole mobile operating system. A large amount of work has tried to face such a challenge head-on by finding and eliminating bugs in existing software using static analysis [77] and model checking [154, 95]. These solutions have an important limitation: they do not scale to large software systems, e.g., the whole Android code base. Moreover, some of these solutions might not be practical as important parts of the I/O stack in mobile operating systems, including Android, are closed source.

Operating systems have employed kernel interpreters for syscall monitoring, e.g., Linux Seccomp, or packet filtering [141]. Jitk [174] provides a verified kernel interpreter using Coq. Our solution in Viola can also be considered as filtering the states and behavior of sensors and indicators in mobile systems. Indeed, our approach in how we use Coq to verify the functional correctness of our compilers has been influenced by Jitk. However, unlike Jitk that performs syscall and network filtering and socket monitoring, Viola monitors the operating system's interactions with I/O devices.

Finally, our verified compiler in Viola and Ditio is related to existing work on verified compilers [126], verified kernels and hypervisors [122, 125, 142, 113, 88], and verified file systems [89]. Indeed, our compiler is built on top of a verified compiler (i.e., CompCert [126]). However, unlike Viola and Ditio, none of these solutions provide formal guarantees about the behavior of I/O devices.

7.4.3 Information Flow Control

Systems such as TaintDroid [102] and Panorama [177] can track the flow of information from sources that produce sensitive information, e.g., camera and microphone, to sinks that can leak the information, e.g., network interface card. Such systems can therefore be used to notify the user when sensor data propagate to sensitive sinks. However, there are key differences between such notifications and Viola. For instance, Viola is designed to reliably detect when sensors are turned on and off using the runtime monitor and device specifications. In contrast, information flow control solutions track the propagation of sensor data and potentially inform the user if the data leave the mobile system. In this sense, Viola and information flow control systems can provide complementary forms of notifications, i.e., notification about when the sensor is on vs. notification about when the sensor data are about to leave the mobile system.

7.4.4 System Verification

In Tabellion, we use formal verification to prove correctness of the checkers we use for analyzing sensor activity logs. Many other systems have used verification tools for building trustworthy systems as well. Examples are a formally verified in-kernel interpreter (Jitk [174]), a certified crash-safe file system (FSCQ [89]), a verified C compiler (CompCert [126]), and verified microkernels and hypervisors (SeL4 [122] and others [125, 142, 113, 88]).

7.4.5 Attacks on Electronic Signatures

Several papers study different forms of attacks on electronic signatures [115, 124]. Dali attack [83, 84, 157] is a confusion attack enabled by the same file being interpreted differently with different file extensions (§6.2). Other forms of attacks use malicious font [119] and JavaScript code [121] as the source of the dynamic representation. Tabellion protects against these attacks as discussed in §6.7.2.

7.4.6 Delay-resistent Systems

TimeSeal [69] designs a secure timer for SGX. It uses counting threads to improve the resolution of SGX timer and addresses the scheduling attacks to the counting threads using policies. It also addresses the delay attacks between the application enclave and the Platform Service Enclave (PSE) that provides the trusted timer in SGX. However, it does not address the NTP delay attacks. Timeline [68] provides a time abstraction for the OS. It includes an assymetric interval that shows time accuracy, however, it does not address delay attacks. cTPM [86] provides secure time for TPM. It uses a trusted cloud server to update the time on the local TPM device. It addresses the delay attack in the NTP by using a global timeout value. However, this technique needs to re-do the synchronization if the delay is more than the timeout value. Sandha et al. [162] evaluates accuracy of time synchronization on smartphones using different hardwares but does not discuss attacks on synchronization.

Chapter 8

Conclusions

In this dissertation, we presented our design and implementation of the security monitor based on the virtualization hardware and TrustZone. We showed that important security services can be built based on the security monitor with small TCB which improves the security of these services. Security monitor's design is low-level, generic, and on-demand. We demonstrated three systems based on the security monitor. First, we presented Viola that provides trustworthy sensor notifications using low-level invariant checks in the security monitor. Second, we presented Ditio that provides trustworthy auditing of sensor activities by recording the sensor activities in the security monitor. Third, we presented Tabellion that provides strong evidence for secure legal contracts by including secure primitives in the security monitor. Finally, we showed that enabling the security monitor adds minimal overhead to the operating system and the extra overhead can be mitigated by leveraging a few techniques in the hypervisor.

Bibliography

[1] Android Detective Video Recorder. https://play.google.com/store/apps/details?id=com.rivalogic.android.video&hl=en.

[2] Android Easy Voice Recorder. https://play.google.com/store/apps/details?id=com.coffeebeanventures.easyvoicerecorder&hl=en.

[3] Android IP Webcam application. https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en.

[4] Android Verified Boot. https://source.android.com/security/verifiedboot/verified-boot.html.

[5] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[6] Dendroid: Android Trojan Being Commercialized. http://blog.trustlook.com/2014/03/20/dendroid-android-trojan-commercialized/.

[7] Disabling the microphone on Google Home (item 11). https://www.cnet.com/how-to/google-home-tips-and-tricks/.

[8] Fuzzing Android System Services by Binder Call to Escalate Privilege. https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege.pdf.

[9] Google Home. https://www.ifixit.com/Teardown/Google+Home+Teardown/72684.

[10] How the NSA can 'turn on' your phone remotely. http://money.cnn.com/2014/06/06/technology/security/nsa-turn-on-phone/.

[11] Introduction to trusted execution environments (tee).

[12] Man spies on Miss Teen USA. http://www.reuters.com/article/2013/10/31/us-usa-missteen-extortion-idUSBRE99U1G520131031.

[13] mbed TLS. https://tls.mbed.org/.

[14] Men spy on women through their webcams. http://arstechnica.com/tech-policy/2013/03/rat-breeders-meet-the-men-who-spy-on-women-through-their-webcams/.

[15] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[16] Open-TEE. https://open-tee.github.io/.

[17] OpenEmbedded. http://www.openembedded.org/.

[18] Panasonic AN30259A LED (used in Galaxy Nexus). http://www.semicon.panasonic.co.jp/ds4/AN30259A_AEB.pdf.

[19] Samsung Gear 2 uses Exynos 3250 SoC. https://www.sammobile.com/2014/04/09/sammobile-confirms-new-exynos-3250-soc-powers-gear-2-gear-fit-uses-cortex-m4-chip/.

[20] The Coq Proof Assistant. https://coq.inria.fr/.

[21] Viola's source code. https://trusslab.github.io/viola/.

[22] Viola's video demo. http://www.ics.uci.edu/~ardalan/viola.html.

[23] ELECTRONIC SIGNATURES IN GLOBAL AND NATIONAL COMMERCE ACT. PUBLIC LAW 106–229, 2000.

[24] ARM Security Technology, Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2004.

[25] NIST Study Shows Computerized Fingerprint Matching Is Highly Accurate. https://www.nist.gov/news-events/news/2004/07/nist-study-shows-computerized-fingerprint-matching-highly-accurate, 2004.

[26] TrustZone: Integrated Hardware and Software Security: Enabling Trusted Computing in Embedded Systems. In *ARM White Paper*, 2004.

[27] *Labajo v. Best Buy Stores, LP*, 478 F. Supp. 2d 523 (S.D.N.Y. 2007)., 2007.

[28] *Adams v. Quicksilver, Inc.* no. G042012 (Cal. App. 4th Div. Feb. 22, 2010), 2010.

[29] An overview of Samsung KNOX. In *Samsung White Paper*, 2013.

[30] Xen on ARM. https://www.slideshare.net/xen_com_mgr/alsf13-stabellini, 2013.

[31] *O'Connor v. Uber Technologies, Inc.*, 150 F.Supp.3d 1095 (N.D. Cal. 2015), 2015.

[32] Fiasco.OC microkernel. https://os.inf.tu-dresden.de/fiasco/, 2016.

[33] *In re Mayfield*: No. 16-22134-D-7, 2016 WL 3958982 (E.D. Cal. July 13, 2016). https://www.govinfo.gov/content/pkg/USCOURTS-caeb-2_16-bk-22134/pdf/USCOURTS-caeb-2_16-bk-22134-0.pdf, 2016.

[34] Trusty TEE. https://source.android.com/security/trusty/, 2017.

[35] Android Protected Confirmation. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html, 2018.

[36] Bluetooth Vulnerability (BlueBorne). https://www.armis.com/blueborne/, 2018.

[37] Broadcom Wi-Fi bug. https://nvd.nist.gov/vuln/detail/CVE-2017-9417, 2018.

[38] Global +$4 Billion Digital Signature Market by Deployment, Component, Industry and Region - Forecast to 2023 - ResearchAndMarkets.com. https://www.businesswire.com/news/home/20181001005761/en/Global-4-Billion-Digital-Signature-Market-Deployment, 2018.

[39] HiKey (LeMaker version) Specification. http://www.lemaker.org/product-hikey-specification.html, 2018.

[40] Open Portable Trusted Execution Environment (OP-TEE). https://www.op-tee.org/, 2018.

[41] 7 landmark electronic signature legal cases. https://esignrecords.org/7-landmark-electronic-signature-legal-cases/, 2019.

[42] AdobeSign. https://acrobat.adobe.com/us/en/sign.html, 2019.

[43] BioID Liveness Detection. https://www.bioid.com/liveness-detection/, 2019.

[44] Checkr. https://checkr.com/product/screenings/, 2019.

[45] Cloudfare Secure Time Service. https://developers.cloudflare.com/time-services/nts/usage/, 2019.

[46] Contract Templates and Agreements. https://www.docsketch.com/contracts/, 2019.

[47] Device-side Security: Samsung Pay, TrustZone, and the TEE. https://developer.samsung.com/tech-insights/pay/device-side-security, 2019.

[48] DocuSign Website. https://www.docusign.com/, 2019.

[49] eSignLive. https://www.esignlive.com/, 2019.

[50] Gesture Recognition. https://github.com/Gogul09/gesture-recognition, 2019.

[51] Global Digital Signature Market to Reach $3.44 Billion by 2022 at 30.0% CAGR: Says AMR. https://www.globenewswire.com/news-release/2019/08/13/1901155/0/en/Global-Digital-Signature-Market-to-Reach-3-44-Billion-by-2022-at-30-0-CAGR-Says-AM html, 2019.

[52] HelloSign. https://www.hellosign.com, 2019.

[53] Intel Provisioning Certification Service for ECDSA Attestation. https://api.portal.trustedservices.intel.com/provisioning-certification, 2019.

[54] iOS Security – iOS 12.3. https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf, 2019.

[55] Measuring Device Power. https://source.android.com/devices/tech/power/device, 2019.

[56] PandaDoc. https://www.pandadoc.com/, 2019.

[57] Qualcomm's larger in-screen fingerprint sensor could seriously improve security. https://www.engadget.com/2019/12/03/qualcomm-3d-sonic-max-worlds-largest-in-display-fingerprint-sensor-specs-availabil 2019.

[58] Samsung Knox Security Solution. https://images.samsung.com/is/content/samsung/p5/global/business/mobile/SamsungKnoxSecuritySolution.pdf, 2019.

[59] Samsung Pay. https://www.samsung.com/us/samsung-pay/, 2019.

[60] SignEasy. https://signeasy.com/, 2019.

[61] SignNow. https://www.signnow.com/, 2019.

[62] Votz. https://voatz.com/, 2019.

[63] Apple Platform Security, Spring 2020. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf, 2020.

[64] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 2006.

[65] A. Amiri Sani. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proc. ACM MobiSys*, 2017.

[66] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proc. ACM SOSP*, 2011.

[67] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against Malicious Peripherals with Cinch. In *Proc. USENIX Security Symposium*, 2016.

[68] F. Anwar, S. D'souza, A. Symington, A. Dongare, R. Rajkumar, A. Rowe, and M. Srivastava. Timeline: An Operating System Abstraction for Time-Aware Applications. In *IEEE Real-Time Systems Symposium (RTSS)*, 2016.

[69] F. M. Anwar. *Quality of Time: A New Perspective in Designing Cyber-Physical Systems*. PhD thesis, UCLA, 2019.

[70] ARM. ARM Cortex-A15 MPCore Processor Technical Reference Manual, Revision: r4p0. *ARM DDI*, 0438I (ID062913), 2013.

[71] ARM. ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual, Revision: r0p1. *ARM DDI*, 0504C (ID063014), 2013, 2014.

[72] ARM. ARM Cortex-A57 MPCore Processor Technical Reference Manual, Revision: r1p3. *ARM DDI*, 0488H, 2013, 2014, 2016.

[73] ARM. Juno ARM Development Platform SoC, Revision r0p0, Technical Overview. *ARM DTO*, 0038A (ID040516), 2014.

[74] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *Proc. USENIX OSDI*, 2016.

[75] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM Trustzone Secure World. In *Proc. ACM CCS*, 2014.

[76] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proc. ACM MobiSys*, 2016.

[77] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[78] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *Proc. USENIX OSDI*, 2014.

[79] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. USENIX OSDI*, 2010.

[80] K. Boos, A. Amiri Sani, and L. Zhong. Eliminating State Entanglement with Checkpoint-based Virtualization of Mobile OS Services. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2015.

[81] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[82] M. Brocker and S. Checkoway. iSeeYou: Disabling the MacBook Webcam Indicator LED. In *Proc. USENIX Security Symposium*, 2014.

[83] F. Buccafurri, G. Caminiti, and G. Lax. The Dalì Attack on Digital Signature. *Journal of Information Assurance and Security*, 2008.

[84] F. Buccafurri, G. Caminiti, and G. Lax. Fortifying the Dalì Attack on Digital Signature. In *Proc. ACM Int. Conf. on Security of Information and Networks (SIN)*, 2009.

[85] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proc. ACM ASPLOS*, 2013.

[86] C. Chen, H. Raj, S. Saroiu, and A. Wolman. cTPM: A cloud TPM for Cross-Device Trusted Applications. In *Proc. USENIX NSDI*, 2014.

[87] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[88] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proc. ACM PLDI*, 2016.

[89] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proc. ACM SOSP*, 2015.

[90] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. ACM ASPLOS*, 2008.

[91] L. Cherkasova and R. Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proc. USENIX ATC*, 2005.

[92] M. A. Chirelstein. *Concepts and Case Analysis in the Law of Contracts, Seventh Edition.* Foundation Press, 2013.

[93] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *Proc. USENIX ATC*, 2016.

[94] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP*, 2001.

[95] J. Croft, R. Mahajan, M. Caesar, and M. Musuvathi. Systematically Exploring the Behavior of Control Programs. In *Proc. USENIX ATC*, 2015.

[96] C. Dall, S. Li, J. T. Lim, J. Nieh, and G. Koloventzos. ARM virtualization: performance and architectural implications. In *Proc. ACM ISCA*, 2016.

[97] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proc. ACM ASPLOS*, 2014.

[98] N. V. Database. Vulnerability summary for cve-2015-6639.

[99] DocuSign. Going Mobile with Electronic Signatures. `https://www.docusign.com/sites/default/files/Going_Mobile_with_Electronic_Signatures.pdf`, 2012.

[100] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proc. IEEE Security and Privacy (S&P)*, 2011.

[101] H. Douglas. *Thin Hypervisor-Based Security Architectures for Embedded Platforms*. PhD thesis, Royal Institute of Technology, 2010.

[102] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. USENIX OSDI*, 2010.

[103] X. Feng and Z. Shao. Modular Verification of Concurrent Assembly Code with Dynamic Thread Creation and Termination. In *Proc. ACM International Conference on Functional Programming (ICFP)*, 2005.

[104] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *Proc. ACM PLDI*, 2006.

[105] Y. Fu and Z. Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proc. IEEE Security and Privacy (S&P)*, 2012.

[106] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proc. USENIX LISA*, 2006.

[107] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Internet Society NDSS*, 2003.

[108] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proc. IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[109] C. Gehrmann, H. Douglas, and D. K. Nilsson. Are there good Reasons for Protecting Mobile Phones with Hypervisors? In *Proc. IEEE Consumer Communications and Networking Conference (CCNC)*, 2011.

[110] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. Toward Trustworthy Mobile Sensing. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2010.

[111] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. YouProve: Authenticity and Fidelity in Mobile Sensing. In *Proc. ACM SenSys*, 2011.

[112] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. ELI: Bare-Metal Performance for I/O Virtualization. In *Proc. ACM ASPLOS*, 2012.

[113] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S. Weng, H. Zhang, and Y. Guo. Deep Specifications and Certified Abstraction Layers. In *Proc. ACM POPL*, 2015.

[114] G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2010.

[115] Hernandez-Ardieta, J. L. and Gonzalez-Tablas, A. I. and de Fuentes, J. M. and Ramos, B. A taxonomy and survey of attacks on digital signatures. *Elsevier Computers & Security*, 2013.

[116] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proc. ACM ASPLOS*, 2013.

[117] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vTZ: Virtualizing ARM TrustZone. In *Proc. USENIX Security Symposium*, 2017.

[118] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 2018.

[119] A. Jøsang, D. Povey, and A. Ho. What you see is not always what you sign. In *Proc. AUUG*, 2002.

[120] A. Kahate. *Cryptography and Network Security.* Tata McGraw-Hill Education, 2013.

[121] K. Kain. Electronic Documents and Digital Signatures. *Master of Science Thesis, Dartmouth Computer Science Department, Technical Report TR2003-457*, 2003.

[122] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SOSP*, 2009.

[123] K. Kostiainen, J. Ekberg, N. Asokan, and A. Rantala. On-board Credentials with Open Provisioning. In *Proc. ACM International Symposium on Information, Computer, and Communications Security (ASIACCS)*, 2009.

[124] G. Lax, F. Buccafurri, and G. Caminiti. Digital Document Signing: Vulnerabilities and Solutions. *Information Security Journal: A Global Perspective*, 2015.

[125] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. International Symposium on Formal Methods (FM)*. Springer, 2009.

[126] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 2009.

[127] W. Li, H. Li, H. Chen, and Y. Xia. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proc. ACM MobiSys*, 2015.

[128] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proc. ACM MobiSys*, 2018.

[129] Y. Li, Y. Li, Q. Yan, H. Kong, and R. H. Deng. Seeing Your Face Is Not Enough: An Inertial Sensor-Based Liveness Detection for Face Authentication. In *Proc. ACM CCS*, 2015.

[130] Y. Li, Z. Wang, Y. Li, R. Deng, B. Chen, W. Meng, and H. Li. A Closer Look Tells More: A Facial Distortion Based Liveness Detection for Face Authentication. In *Proc. ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2019.

[131] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proc. USENIX Security Symposium*, 2016.

[132] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software Abstractions for Trusted Sensors. In *Proc. ACM MobiSys*, 2012.

[133] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. USENIX ATC*, 2006.

[134] M. Liu, T. Li, N. Jia, A. Currid, and V. Troy. Understanding the Virtualization "Tax" of Scale-out Pass-Through GPUs in GaaS Clouds: An Empirical Study. In *Proc. IEEE HPCA*, 2015.

[135] N. Lynch and F. Vaandrager. Forward and Backward Simulations Part I: Untimed Systems. *Information and Computation*, 1995.

[136] N. Lynch and F. Vaandrager. Forward and Backward Simulations Part II: Timing-Based Systems. *Information and Computation*, 1996.

[137] Z. Ma, S. Mirzamohammadi, and A. Amiri Sani. Understanding Sensor Notifications on Mobile Devices. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2017.

[138] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My VM is Lighter (and Safer) than your Container. In *Proc. ACM SOSP*, 2017.

[139] S. Matetic, M. Schneider, A. Miller, A. Juels, and S. Capkun. DelegaTEE: Brokered Delegation Using Trusted Execution Environments. In *Proc. USENIX Security*, 2018.

[140] E. Maxie. COURT CASE: LAWSUIT FILED OVER POORLY CONCEIVED ELECTRONIC SIGNATURE. `https://www.signix.com/blog/bid/93126/court-case-lawsuit-filed-over-poorly-conceived-electronic-signature`, 2013.

[141] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proc. Winter 1993 USENIX Technical Conference*, 1993.

[142] M. McCoyd, R. B. Krug, D. Goel, M. Dahlin, and W. Young. Building a Hypervisor on a Formally Verifiable Protection Layer. In *Proc. IEEE Hawaii International Conference on System Sciences (HICSS)*, 2013.

[143] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. USENIX ATC*, 1996.

[144] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *Proc. USENIX OSDI*, 2000.

[145] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing Speech from Gyroscope Signals. In *Proc. USENIX Security*, 2014.

[146] R. Mijat and A. Nightingale. Virtualization is Coming to a Platform Near You. ARM White Paper, 2011.

[147] S. Mirzamohammadi and A. Amiri Sani. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. In *Proc. ACM MobiSys*, 2016.

[148] S. Mirzamohammadi and A. Amiri Sani. The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2018.

[149] S. Mirzamohammadi and A. Amiri Sani. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. *IEEE Transactions on Mobile Computing (TMC)*, 2018.

[150] S. Mirzamohammadi, J. A. Chen, A. Amiri Sani, S. Mehrotra, and G. Tsudik. Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices. In *Proc. ACM SenSys*, 2017.

[151] S. Mirzamohammadi, Y. Liu, T. A. Huang, A. Amiri Sani, S. Agarwal, and S. E. Kim. Tabellion: System Support for Secure Legal Contracts. In *Proc. ACM MobiSys*, 2020.

[152] J. Mulliner. What the Wells Fargo Mobile Research Reveals About E-Signatures. `https://www.onespan.com/blog/what-the-wells-fargo-mobile-research-reveals-about-e-signatures`, 2018.

[153] I. Muslukhov, Y. Boshmaf, C. Kuo, J. Lester, and K. Beznosov. Understanding Users' Requirements for Data Protection in Smartphones. In *Proc. IEEE Int. Conf. on Data Engineering Workshops (ICDEW)*, 2012.

[154] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proc. USENIX OSDI*, 2002.

[155] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 1978.

[156] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten Years Later. In *Proc. ACM ASPLOS*, 2011.

[157] D. Popescu. Hiding Malicious Content in PDF Documents. *arXiv preprint arXiv:1201.0397*, 2012.

[158] R. S. Portnoff, L. N. Lee, S. Egelman, P. Mishra, D. Leung, and D. Wagner. Somebody's Watching Me? Assessing the Effectiveness of Webcam Indicator Lights. In *Proc. ACM CHI*, 2015.

[159] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16), Austin, TX*, 2016.

[160] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. BootStomp: On the Security of Bootloaders in Mobile Devices. In *Proc. USENIX Security Symposium*, 2017.

[161] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-Guided Device Driver Synthesis. In *Proc. USENIX OSDI*, 2014.

[162] S. S. Sandha, J. Noor, F. M. Anwar, and M. Srivastava. Exploiting Smartphone Peripherals for Precise Time Synchronization. In *Proc. IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, 2019.

[163] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. ACM ASPLOS*, 2014.

[164] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proc. ACM VEE*, 2009.

[165] D. Silva. Demand for E-Signing From Mobile Devices on the Rise in Financial Institutions. `https://www.onespan.com/blog/demand-for-e-signing-from-mobile-devices-on-the-rise-in-financial-institutions`, 2019.

[166] M. Simpson. BDC app offers e-signature for loans, reducing in-person visits. `https://www.itbusiness.ca/news/bdc-app-offers-e-signature-for-loans-reducing-in-person-visits/104429`, 2018.

[167] M. H. Stanzione. 'Wet' Ink Signatures Requirements May Fade After Coronavirus. Bloomberg Law, The United States Law Week, 2020.

[168] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proc. ACM EuroSys*, 2010.

[169] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. TrustDump: Reliable Memory Acquisition on Smartphones. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2014.

[170] D. Tang, Z. Zhou, Y. Zhang, and K. Zhang. Face Flashing: a Secure Liveness Detection Protocol based on Light Reflections. *arXiv preprint arXiv:1801.01949v2*, 2018.

[171] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proc. USENIX Security Symposium*, 2018.

[172] M. Vanhoef and F. Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proc. ACM CCS*, 2017.

[173] W. Wang, Z. Shao, X. Jiang, and Y. Guo. A Simple Model for Certifying Assembly Programs with First-Class Function Pointers. In *Proc. IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2011.

[174] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: a Trustworthy In-Kernel Interpreter Infrastructure. In *Proc. USENIX OSDI*, 2014.

[175] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proc. USENIX OSDI*, 2008.

[176] J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *Proc. ACM Workshop on Scalable Trusted Computing (STC)*, 2008.

[177] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. ACM CCS*, 2007.

[178] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du. TruZ-Droid: Integrating TrustZone with Mobile Operating System. In *Proc. ACM MobiSys*, 2018.

[179] K. Ying, P. Thavai, and W. Du. Truz-view: Developing trustzone user interface for mobile os using delegation integration model. In *Proc. ACM CODASPY*, 2019.

[180] D. Yu and Z. Shao. Verification of Safety Properties for Concurrent Assembly Code. In *Proc. ACM International Conference on Functional Programming (ICFP)*, 2004.

[181] M. Yu, V. D. Gligor, and Z. Zhou. Trusted Display on Untrusted Commodity Platforms. In *Proc. ACM CCS*, 2015.

[182] D. Zhang. TrustFA: TrustZone-Assisted Facial Authentication on Smartphone. *Technical Report*, 2014.

[183] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *Proc. ACM CCS*, 2019.

[184] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2012.