REGULATING SMART DEVICES IN RESTRICTED SPACES

By

DAEYOUNG KIM

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Vinod Ganapathy

And approved by

____

____

____

____

New Brunswick, New Jersey

January, 2019

ABSTRACT OF THE DISSERTATION

Regulating Smart Devices in Restricted Spaces

by DAEYOUNG KIM

Dissertation Director:
Vinod Ganapathy

Smart devices have spread everywhere in our daily lives, and the capabilities of smart devices equipped with a variety of sensors and peripherals are increasing. However, these devices can possibly be misused in various environments. For instance, sensitive data in enterprises and federal offices can be leaked by the use of cameras and microphones on smart devices. In classrooms, students can obtain unauthorized information during exams. Moreover, smart devices can be used to take pictures or record videos without permissions in less stringent environments such as gym locker rooms and movie theaters. Therefore, we need methods to prevent these situations instead of ad hoc methods in such restricted spaces.

In this dissertation, we focus on how to regulate smart devices in restricted spaces. We propose ARM TrustZone-based solutions to enforce policies on smart devices. In particular, the dissertation makes the following two contributions.

First, we present a systematic approach for restricted space hosts to analyze and regulate guest devices using remote memory operations in the restricted space. In our approach, hosts' policies are enforced by a small trusted computing base that executes on the guest devices. We also show that our approach provides strong security guarantees by leveraging the ARM TrustZone.

Second, we propose ForceDroid, a policy enforcement system that provides a higher-level abstraction in the restricted spaces. We leverage Security-Enhanced Linux in Android (SE-Android) to support fine-grained access control, and use Near field communication (NFC) to securely communicate between guests and hosts. In ForceDroid, predefined policies on guest devices are enforced by hosts' requests.

Acknowledgements

First of all, I would like to thank my graduate adviser Professor Vinod Ganapathy, for his great insight and valuable advice. I am extremely grateful to him for believing in me and supporting me whenever I struggled during my graduate journey. I would also like to express my deepest gratitude to Professor Liviu Iftode, who is resting in peace, for his co-advising. He had so much passion and gave it to me on my research and studies. I will miss him dearly.

I would like to extend my sincere thanks to my committee members, Professor Badri Nath, Professor Abhishek Bhattacharjee, and Dr. Pratyusa Manadhata for their helpful advice. I would also like to extend my gratitude to my collaborators, Ferdinand Brasser, Christopher Liebchen, Dr. Ahmad-Reza Sadeghi from Technische Universitat Darmstadt, and Dr. Abhinav Srivastava, for their contributions to the joint research projects.

I would like to thank the past members of Disco lab, Liu Yang, Mohan Dhawan, Lu Han, Wenjie Sha, Hongzhang Liu, Nader Boushehrinejadmoradi, Daehan Kwak, Amruta Gokhale, Shakeel Butt, Rezwana Karim, Ruilin Liu, Hai Nguyen. I still remember that they gave me a warm welcome when I came to Rutgers first. Also, many thanks especially to Daehan Kwak, Amruta Gokhale, Shakeel Butt, Rezwana Karim, Ruilin Liu, Hai Nguyen, for their helpful advice and support. Lastly, I am deeply indebted to my family for their unconditional love, patience, and support throughout this entire journey.

Dedication

To my family, for their endless support and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

This dissertation describes two approaches for regulating smart devices with ARM TrustZone in restricted spaces. The first approach is regulating smart devices with remote memory operations, and the second approach is enforcing policies on smart devices with Security-Enhanced Linux in Android (SEAndroid).

1.1 Motivation

Recently, we have observed a rapid increase in the number and a diversity of smart devices. These include smartphones, tablets, watches, and home devices. These smart devices have become integrated into our daily lives. As a result of these developments, smart devices will be used in a wide variety of environments, such as the home, in social settings, at school and work. We bring the smart devices as we go, and anticipate that the devices connect and work with the settings at the places we visit.

As smart devices evolve, their computing power will have increased, and the quality and a diversity of peripherals will be available on them. While this is generally a desirable trend, highly capable smart devices can be misused to exfiltrate sensitive data or obtain unauthorized information in some restricted environments. This may happen either through overt malice by the device owner, or unintentionally, via malicious apps accidentally installed on the device. Examples of such environments abound in today's society, and they impose a wide variety of policies (often unwritten) governing the use of smart devices. Enterprises typically forbid employees' personal devices from connecting to corporate networks or storing corporate data. Federal institutions and laboratories that process sensitive information place even more stringent restrictions, often requiring employees and visitors to place personal devices in Faraday cages before entering certain areas. In the classroom setting, students are often forbidden from using the aid of smart devices during examinations. Even outside work and school, there may be privacy concerns that restrict how smart devices are used. For example, certain restaurants and bars ban patrons from wearing smart glasses [56]. In social settings, people may be uncomfortable at the thought of their conversations being recorded by smart devices. We use the term restricted spaces to refer to such environments.

The mechanisms traditionally used to regulate device use in restricted spaces are ad hoc. Consider, for instance, the restricted spaces discussed above. In the enterprise setting, employees are given a separate work laptop/phone that can connect to the enterprise network. They are also implicitly, or by contract, forbidden from copying data from these corporate devices onto their personal devices. In the federal setting, employees and visitors must undergo physical security checks to ensure that they are not carrying electronic equipment, while in the classroom setting, proctors enforce compliance. Moreover, in social settings, enforcement is informal and left to privacy-conscious patrons or hosts.

Going forward, such ad hoc methods will be inadequate. First, our growing reliance on smart devices will make traditional methods challenging to use. For example, employees or students may also wish to use assistive devices for legitimate reasons such as smart glasses or contacts lens for vision correction, Bluetooth-enabled hearing aids, and smart watches to monitor time. It would not be possible to ask them to refrain from using the devices. In this case, the right solution would be to permit the devices, but regulate the use of its camera, microphone or WiFi interfaces. Second, the increasing diversity of smart devices will make it hard to identify policy violations (even if the devices are not used covertly). For example, there is already evidence that students cheat on exams by connecting to the Internet using smart phones [7]. Recent research has demonstrated even subtler methods to outsmart proctors via the use of smartwatches [59]. Even in less strict social settings, such as restaurants, conferences, gym locker rooms and private homes, smart devices may be used to take photos and videos, or record private conversations that could invade privacy. And third, in enterprise settings, current trends indicate that employees may be encouraged to use their smart devices for corporate purposes. The Bring Your Own Device (BYOD) trend has numerous benefits, such as device consolidation and reducing the enterprise's cost of device procurement. With BYOD, it is imperative to ensure that employee devices adhere to corporate policies within the enterprise.

Given these reasons, we posit that a systematic method is needed to ensure that smart personal devices comply with the policies of the restricted spaces within which they are used. The hosts and the guests will benefit from such a method. The hosts will acquire greater assurance that smart devices used in their spaces comply with their policies. Additionally, the guests can be more open about their smart devices use in the hosts' restricted space. Therefore, we not only require to regulate smart devices, but also need systematic methods for ensuring responsible use.

This dissertation is divided into two parts. In the first part, we focus on how to regulate smart devices with remote memory operations. In the second part, we describe how to regulate smart devices with SEAndroid.

1.2 Regulating Smart Devices with Remote Memory Operations

We present an approach for hosts to remotely regulate the use of smart guest devices in restricted spaces. Our approach recognizes that policies to govern device use vary widely across restricted spaces, and therefore cleanly separates policy from mechanism. Policies are specified by hosts and could, for instance, require guests to prove that their devices are free of certain forms or malicious software, or restrict the use of certain peripherals on the devices, e.g., camera, WiFi, or 3G/4G.

The enforcement mechanism itself is implemented on the guests' device(s) and simply enforces the hosts' policies. This mechanism must have three key properties. First, it must be trusted by both the host and the guest, and is, therefore, part of the trusted computing base (TCB). The host trusts the mechanism to correctly enforce policies on the guest device. On the other hand, the guest trusts the mechanism to authenticate the host. Host authentication is important because malicious or untrusted hosts could otherwise abuse the mechanism to compromise the guest's security and privacy. Second, the mechanism must have the ability to inspect the guest's device and make changes to its configuration to enforce the hosts' policies. And third, the mechanism must be minimal, and not bloat the size of the TCB executing on the guest devices.

We describe an approach to implement this enforcement mechanism using remote memory operations on the guest device. Hosts use these operations to remotely inspect and modify the memory contents of the guest device based on its policies. The host must be able to ensure that these operations are performed correctly on the guest device. Thus, we cannot rely on the guest's operating system to enforce these operations because it may be malicious (or compromised by malware). We therefore rely on trusted hardware in the guest's device to provide a root of trust, using which hosts can obtain such assurances. In particular, our prototype assumes that the guest device is equipped with the ARM TrustZone architecture [5], which contains a set of hardware features that allows a device to execute trusted code in an isolated partition. ARM TrustZone-enabled devices are now commercially available, with many millions of devices deployed already [11]. We rely on the isolated partition in an ARM TrustZone-enabled device to bootstrap trust in our mechanism's TCB, which executes on the guest device.

1.3 Regulating Smart Devices with SEAndroid

We present an approach to regulate guest devices use with fine-grained access control in restricted spaces. Our previous work focused on the low-level mechanism for policy enforcement. As a result, we built a small TCB that approximately has less than 1000 LoCs and can detect rootkits. Nevertheless, there are shortcomings of our previous work [19]. Firstly, the previous approach could be too intrusive because a guest transmits the image of the kernel memory on the guest device to a host so that privacy problems may occur. Moreover, we do not need a vetting service in our system. Secondly, remote memory operations in the system perform at low-level; thus the previous approach could be complicated to not only deploy to devices, but also maintain and configure it as a vendor's perspective. System administrators need to comprehend the specific memory addresses of each device. Thirdly, the policy language in the approach is not user-friendly since the system reads and writes of memory at low-level, which is best suited to peripheral control, but app or file level access control. Hence, we build a higher-level abstraction scheme to provide fine-grained access control as an alternative approach to our previous work.

We propose a secure policy enforcement scheme called ForceDroid that can dynamically enforce policies on guest devices. The basic idea is to securely apply the SEAndroid policies to guest devices via NFC communications. In our work, we employ SEAndroid for fine-grained access control. We also recognize guest devices' location information from NFC, and leverage the ARM TrustZone on smart devices for isolating the enforcement mechanism and bootstrapping its security features.

Our approach can prevent guests from taking unauthorized videos or photos. Before entering public restricted spaces, some peripherals can be disallowed by using a simple NFC tap. For example, the video recording function of smart devices to record movies must be forbidden in movie theaters, and all camera functions must be prohibited in the locker room at a fitness center. Guests can check hosts' policies on their smart devices and comply with the policies.

In enterprise settings, mobile device management (MDM) solutions for BYOD can be applied to guest devices. However, guest devices may need more permissions in an only specific meeting room. In our approach, guest devices can simply acquire or lose some functions through NFC in front of doors. We built a policy enforcement prototype for not only the work environments, but also public restricted spaces, such as gym locker rooms and movie theaters. Our approach can be easily adopted on NFC-enabled smart devices for policy enforcement.

1.4 Summary of Contributions

Thesis statement: We can leverage ARM TrustZone devices to build methods to regulate smart devices and ensure responsible use in restricted spaces.

This dissertation supports the above thesis statement, and makes the following contributions:

- We introduce the restricted space model for smart personal devices. This model is of independent interest due to the growing number of environments that restrict smart device use. We also present multiple examples of policies that may be enforced in restricted spaces.

- We present the design of a mechanism for hosts to enforce policies on guest device use(Chapter 2). Our mechanism allows hosts to remotely inspect and control a guest device using remote memory operations. The mechanism allows hosts to request a proof from guest devices that they are compliant with hosts' policies. The mechanism ensures that only authenticated hosts can control guest devices, and provides a vetting service that allows guests to check the safety of the host's requests.

- We propose a policy enforcement system that dynamically enforces SEAndroid policies on guest devices in restricted spaces (Chapter 3). In our approach, the system loads predefined policies from guest devices to solve privacy concerns. We provide practical fine-grained access control mechanism by leveraging SEAndroid and NFC. This design allows us to more efficiently configure and maintain smart devices usage policies.
- We demonstrate the prototype implementations of our approaches using the ARM Trust-Zone hardware as a root of trust on guest devices.

1.5 Contributors to the Dissertation

The following is a list of people who co-authored papers from which material was used in this dissertation. Chapter 2 of this dissertation is the result of a collaboration with my adviser, Professor Vinod Ganapathy, Professor Liviu Iftode, my colleagues, Ferdinand Brasser, Christopher Liebchen, and Dr. Ahmad-Reza Sadeghi from Technische Universitat Darmstadt. Professor Vinod Ganapathy was also contributed to the design of the project in Chapter 3.

Chapter 2

Regulating Smart Devices with Remote Memory Operations

This chapter describes an overview of the restricted space model. We present an approach on how to regulate smart devices using remote memory operations in restricted spaces.

2.1 Introduction

Personal computing devices, such as phones, tablets, glasses, watches, assistive health monitors and other embedded devices have become an integral part of our daily lives. We carry these devices as we go, and expect them to connect and work with the environments that we visit.

While the increasing capability of smart devices and universal connectivity are generally desirable trends, there are also environments where these trends may be misused. In enterprise settings and federal institutions, for instance, malicious personal devices can be used to exfiltrate sensitive information to the outside world. In examination settings, smart devices may be used to infiltrate unauthorized information [7], surreptitiously collude with peers [59] and cheat on the exam. Even in less stringent social settings, smart devices may be used to record pictures, videos or conversations that could compromise privacy. We therefore need to regulate the use of smart devices in such *restricted spaces*.

Society currently relies on a number of ad hoc methods for policy enforcement in restricted spaces. In the most stringent settings, such as in federal institutions, employees may be required to place their personal devices in Faraday cages and undergo physical checks before entering restricted spaces. In corporate settings, employees often use separate devices for work and personal computing needs. Personal devices are not permitted to connect to the corporate network, and employees are implicitly, or by contract, forbidden from storing corporate data on personal devices. In examination settings, proctors ensure that students do not use unauthorized electronic equipment. Other examples in less formal settings include restaurants that prevent patrons from wearing smart glasses [56], or privacy-conscious individuals who may request owners to refrain from using their devices.

We posit that such ad hoc methods alone will prove inadequate given our increasing reliance on smart devices. For example, it is not possible to ask an individual with prescription smart glasses (or any other assistive health device) to refrain from using the device in the restricted space. The right solution would be to allow the glass to be used as a vision corrector, but regulate the use of its peripherals, such as the camera, microphone, or WiFi. A general method to regulate the use of smart devices in restricted spaces would benefit both the *hosts* who own or control the restricted space and *guests* who use smart devices. Hosts will have greater assurance that smart devices used in their spaces conform to their usage policies. On the other hand, guests can benefit from and be more open about their use of smart devices in the host's restricted space.[1]

Prior research projects (*e.g.,* [40, 20, 58, 74, 65, 73]) and enterprise mobile-device management (MDM) solutions to address this problem (*e.g.,* Samsung Knox [78], Microsoft Intune [57] and Blackberry EMM [16]) have typically assumed that guest devices are benign. These solutions outfit the guest device with a security-enhanced software stack that is designed to accept and enforce policies supplied by restricted space hosts. A host must trust the software running on a guest device to correctly enforce its policies, and generally has no means to obtain guarantees that a guest device is policy-compliant. Clearly, malicious guest devices with a suitably-modified software stack can easily bypass policy enforcement.

Our vision is to enable restricted space hosts to enforce usage policies on guest devices with provable security guarantees. Simultaneously, we also wish to reduce the amount of trusted policy-enforcement code (*i.e.,* the size of the security-enhanced software stack) that needs to execute on guest devices. To that end, this paper offers a number of advances:

(**1**) **Use of trusted hardware.** We leverage the ARM TrustZone [5] on guest devices to offer provable security guarantees. In particular, a guest device uses the ARM TrustZone to produce *verification tokens*, which are unforgeable cryptographic entities that establish to a host that the guest is policy-compliant. Malicious guest devices, which may have violated the host's

[1]We only consider overt use of guest devices. Covert use must still be addressed using other methods, such as physical checks.

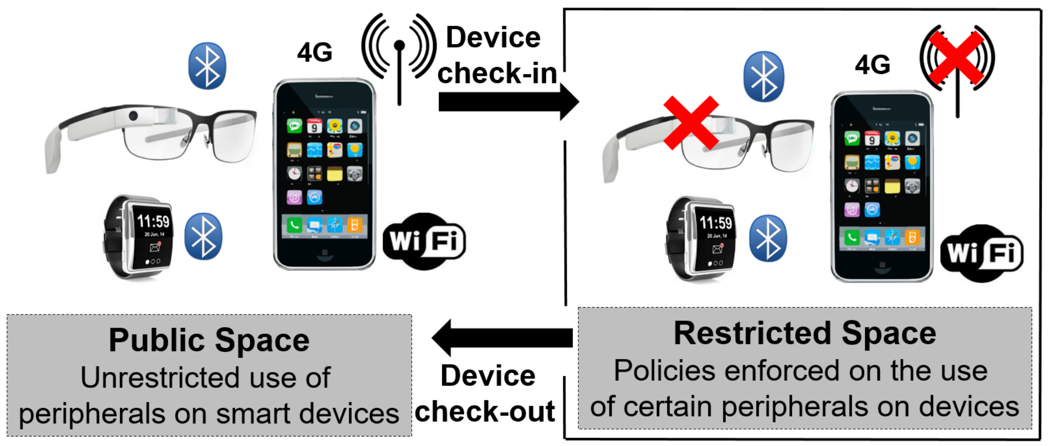
policies in the restricted space, will not be able to provide such a proof, and can therefore be apprehended by the host. Devices that use the ARM TrustZone are now commercially available and widely deployed [11], and our approach applies to these devices.

(**2**) **Remote memory operations.** We use host-initiated remote memory operations as the core method to regulate guest devices. In this approach, a host decides usage policies that govern how guest devices must be regulated within the restricted space. For example, the host may require certain peripherals on the guest device (*e.g.,* camera, WiFi or 3G/4G) be disabled in the restricted space. The host sends these policies to the guest device, where a trusted policy-enforcement mechanism applies these policies by reading or modifying device memory.

The principal benefit of using remote memory operations as an API for policy enforcement is that it considerably simplifies the design and implementation of the policy-enforcement mechanism, while still offering hosts fine-grained control over guest devices. Remote memory operations also give hosts that use our approach the unique ability to scan guest devices for kernel-level malware. Combined with the ARM TrustZone, which helps bootstrap trust in the guest's policy-enforcement code, our approach offers hosts an end-to-end assurance that guest devices are policy-compliant.

(**3**) **Secure device checkpointing.** The downside to enforcing policies by modifying device memory is that changes to the guest device are ephemeral, and can be undone with a simple reboot of the guest device. We therefore introduce *REM-suspend*, a secure checkpointing scheme to ensure that a guest device remains "tethered" to the host's policies even across device reboots and other power-down events.

(**4**) **Vetting for guest device privacy and security.** The advances above benefit hosts, but guests may be uncomfortable with the possibility of hosts accessing and modifying raw memory on their devices. If access to raw guest device memory is not mediated, malicious hosts may be able to use this access to compromise the guest's privacy and security. For example, the host can read sensitive and private app data from devices and install malicious snooping software (*e.g.,* keyloggers) on the guest device. We therefore mediate the host's access to the guest device by introducing a vetting service. The vetting service is trusted and configurable by guests, and allows them to check the safety of the host's memory operations before performing


4G
Device check-in
4G
Public Space
Unrestricted use of peripherals on smart devices
Device check-out
Restricted Space
Policies enforced on the use of certain peripherals on devices

Restricted space model. Guests "check-in" their personal devices when entering restricted spaces. During check-in, hosts inspect, analyze and modify the configurations of these devices in accordance with their usage policies. In this example, the host restricts the use of the camera on the smart glass, and the 4G data interface on the smart phone. However, the glass and watch can continue to use Bluetooth pairing, while the phone can connect to the host's access points using WiFi. When guests leave the restricted space, they "check-out" their devices, restoring them to their original configurations.

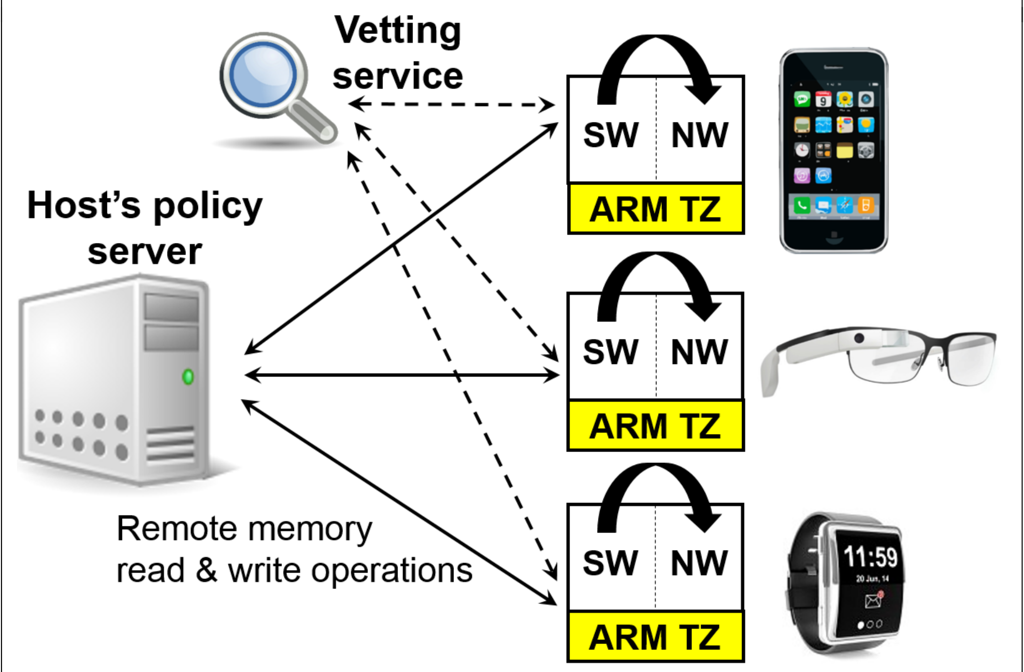
Figure 2.1: An overview of the entities of our restricted space model.

them on the devices. The vetting service ameliorates guests' privacy and security concerns and restricts the extent to which hosts can control their devices.

We built and evaluated a prototype to show the benefits of our approach. We show that a small policy-enforcing code base running on guest devices offers hosts fine-grained policy-based control over the devices. We also show that a vetting service with a few simple sanity checks allows guests to ensure the safety of the host's remote memory operations.

2.2 Restricted Spaces

We provide an overview of the restricted space model, motivate some features of our enforcement mechanism, and describe our threat model. Because our mechanism relies on the ARM TrustZone, we begin by introducing its features.



Guest device setup. Guest devices are equipped with ARM TrustZone and execute components of the policy enforcement mechanism in the secure world (SW). The details of this mechanism appear in §2.4. At check-in, the host's policy server leverages the secure world to remotely inspect and modify normal world (NW) memory.

Figure 2.2: An overview of the setup of guest devices.

2.2.1 Background on the ARM TrustZone

The TrustZone is a set of security enhancements to chipsets based on the ARM architecture. These enhancements cover the processor, memory and peripherals. With TrustZone, the processor executes instructions in one of two security modes at any given time, a *normal world* and a *secure world*. A third *monitor mode* facilitates switching between the normal and the secure worlds. The secure and normal worlds have their own address spaces and different privileges. The processor switches from the normal world to the secure world via an instruction called the secure monitor call (smc). When an smc instruction is invoked from the normal world, the processor context switches to the secure world (via monitor mode) and freezes execution of the normal world.

The ARM TrustZone partitions memory into two portions, with one portion being exclusively reserved for the secure world. It also allows individual peripherals to be assigned to the secure world. For these peripherals, hardware interrupts are directly routed to and handled by the secure world. While the normal world cannot access peripherals or memory assigned to the secure world, the secure world enjoys unrestricted access to all memory and peripherals on the device. It can therefore access the code and data of the normal world. The secure world can execute arbitrary software, ranging from simple applications to an entire operating system (OS).

A device with ARM TrustZone boots up in the secure world. After the secure world has initialized, it switches to the normal world and boots the OS there. Most TrustZone-enabled devices are configured to execute a *secure boot* sequence that incorporates cryptographic checks into the secure world boot process [5, §5.2.2]. For example, the device vendor signs the code with its private key, and the vendor's code in the boot ROM verifies this signature using the vendor's public key. These checks ensure that the integrity of the boot-time code in the secure world has not been compromised, *e.g.,* by reflashing the image on persistent storage. Most vendors lock down the secure world via secure boot, thereby ensuring that it cannot be modified by end-users. This feature allows hosts to trust software executing in the secure world and treat it as part of the trusted computing base (TCB). In this paper, we assume that guest devices use secure boot.

2.2.2 Entering and Exiting Restricted Spaces

Check-in. When a guest enters a restricted space, he checks in each of his devices during entry (Figure 2.1). During check-in, the guest device communicates with the host's policy server for the following tasks:

(**1**) **Authentication.** The first step is for the host and the guest to mutually identify each other. We assume that both the guest and the host have cryptographic credentials (*e.g.,* public/private key pairs) that are validated via a trusted third party, such as a certifying authority. The host and the guest mutually authenticate each other's credentials in the standard way, as is done during SSL/TLS handshakes.

The host's policies are enforced by a mechanism that executes in the secure world of the guest device. We rely on TrustZone's secure boot sequence to prevent unauthorized modifications to this code. Note that the end-user's usual work environment on the device, *e.g.,* the traditional

Android, iOS or Windows environment with apps, executes in the normal world (and is untrusted). We expect the secure world software running the mechanisms proposed in this paper to be created and distributed by trusted entities, such as device vendors, and execute in isolation on guest devices.

(**2**) **Host remotely reads guest state.** The host requests the guest device for a snapshot of its normal world memory and CPU register state. The secure world on the device fulfills this request (after it has been cleared by the vetting service) and sends it to the host. The secure world also sends a cryptographic checksum of this data to prevent unauthorized modifications during transit.

The host uses raw memory pages from the device in two ways. First, it scans memory pages to ensure that the normal world kernel is free of malicious software. A clean normal world kernel can bootstrap additional user-level security mechanisms, *e.g.,* an antivirus to detect malicious user-level apps. Second, it extracts the normal world's configuration information. This includes the kernel version, the list of peripherals supported, memory addresses of various device drivers for peripherals and the state of these peripherals *e.g.,* whether a certain peripheral is enabled and its settings. The host can also checkpoint the configuration for restoration at check-out.

The host terminates check-in at this point if it finds that the guest device is malicious or runs a kernel version that it cannot reconfigure. The action that the host takes depends upon the specific setting. For example, in a federal building, the device owner may be asked to quarantine the device outside the restricted space or enclose it in a physically-secured Faraday cage. In less stringent settings, the host may blacklist the device's MAC address and prevent it from connecting to any local resources in the restricted space. Note that benign end-users may not have willingly installed malware on their devices. A failed check-in has the desirable side-effect of allowing such end-users to detect that their device is infected.

(**3**) **Host remotely modifies guest state.** The host modifies the guest device to conform to its restricted usage policies. The host's restrictions on a guest device depend on what it perceives as potential risks. Cameras and microphones on guest devices are perhaps the most obvious ways to violate the host's confidentiality because they can be used to photograph confidential documents or record sensitive meetings. Networking and storage peripherals such as WiFi, 3G/4G, Bluetooth and detachable storage dongles can work in concert with other peripherals to exfiltrate sensitive information. Dually, guest devices can also be used to infiltrate unauthorized information into restricted spaces, *e.g.,* students can cheat on exams by using their devices to communicate with the outside world.

The host controls peripherals on guest devices by creating a set of updates to the device's normal world memory and requesting the secure world to apply them. For example, one way to disable a peripheral is to unlink its driver from the device's normal world kernel (details in §2.3). The secure world applies these updates after using the vetting service to ensure the safety of the requested updates.

We assume that it is the host's responsibility to ensure that the memory modifications are not easily bypassable. For example, they may be undone if the user of the guest device directly modifies kernel memory, *e.g.,* by dynamically loading kernel modules or using /dev/kmem in the normal world. The host must inspect the guest device's snapshot for configurations that lead to such attacks, and disallow the use of such devices in the restricted space.

In steps (2) and (3), the secure world performs the host's read and write operations only if they are approved by the vetting service. Guests configure the vetting service to suit their security and privacy goals. If the vetting service deems an operation unsafe, device check-in is aborted and the device is left unmodified. The guest cannot use the device in the restricted space because its security and privacy goals conflict with the host's usage policies.

(**4**) **Host obtains verification token from guest.** After the guest device state has been modified, its secure world produces a verification token to be transmitted to the host. The verification token is a cryptographic checksum over the memory locations that were modified. The token is unforgeable in that only the secure world can re-create its value as long as the host's memory updates have not been altered, and any malicious attempts to modify the token can be detected by the secure world and the host.

The check-in steps above bear some resemblance to TPM-based software attestation protocols developed in the trusted computing community [77]. Like TPM measurements, which attest the software stack or properties of dynamic data structures [76], verification tokens attest that a guest device's state complies with the host's policies. Both TPM measurements and verification tokens are grounded in a hardware root of trust. However, unlike traditional software attestation, which has largely been restricted to passive checks of a remote machine's state, verification tokens attest to the integrity of the host's remote modifications of the guest device. Like in software attestation, a correct verification token attests the state of the guest device only at the instant at which it was produced by the secure world. To ensure that the guest device remains policy-compliant, the host can request the device to send it the verification token at any point when the device is in the restricted space. The secure world on the device computes this token afresh, and transmits it to the host,[2] which compares this freshly-computed token with the one obtained during check-in. It uses this comparison to ensure that the guest has not altered the normal world memory updates from the previous step. The verification token incorporates a host-supplied challenge to ensure that the guest device cannot simply replay old tokens. As we demonstrate in §2.6, verification tokens are only a few hundred bytes in size and can be computed by the secure world in just a few milliseconds. Thus, hosts can request guest devices to send verification tokens at frequent intervals, thereby increasing confidence that the guest device was continuously policy-compliant.

The verification token is ephemeral, and can be computed afresh by the guest only within an expiration period. The token expires upon device check-out or if the device is powered off, thereby ensuring that end-users cannot undo the host's memory updates by simply rebooting the device. In §2.4.4, we describe *restricted space-mode (REM) suspend*, a special protocol that suspends the device while allowing the verification token and the host's memory updates to persist.

Check-out. Once checked-in, the guest device are free to avail of the facilities of the restricted space under the policies of the host. For example, in Figure 2.1, the smart glass and watch can pair with the smart phone via Bluetooth, while the smart phone can use the host's WiFi access point. When the guest checks-out, two tasks must be accomplished:

(**1**) **Host checks guest state.** The host requests the guest to send the verification token to ensure that the device is policy-compliant. The token may not match the value obtained from

[2]This assumes that the host's policy still allows communication between the host and the guest. If all of the guest's peripherals are disabled, the host must physically access the guest to visually obtain the fresh token.

the device at check-in if the host's memory modifications have been maliciously altered or if the end-user chose to consciously bypass REM-suspend and reboot the device. It is not possible to differentiate between these cases, and the host's policy to deal with mismatches depends upon the sensitivity of the restricted environment. For example, in a federal setting, detailed device forensics may be necessary. As previously discussed, hosts can request the verification token from the device at any time when it is in the restricted space. Hosts use this feature to frequently check the verification token to narrow the timeframe of the violation.

(**2**) **Restoring guest state.** To restore the state of the device, the end-user simply performs a traditional device reboot. The host only modifies the memory of the device, and not persistent storage. Rebooting therefore undoes all the memory modifications performed by the host and boots the device from an unmodified version of the kernel in persistent storage. Alternatively, the host can restore the state of the device's peripherals from a checkpoint created at check-in. The main challenge here is to ensure consistency between the state of a peripheral and the view of the peripheral from the perspective of user-level apps. For example, when the 3G interface is disabled, an app loses network connectivity. However, because we only modify memory and do not actually reset the peripheral, the 3G card may have accumulated packets, which the app may no longer be able to process when the kernel state is restored. Mechanisms such as shadow drivers [89] can possibly enable such "hot swaps" of kernel state and avoid a device reboot.

2.2.3 Threat Model

We now summarize our threat model. From the host's perspective, the guest device's normal world is untrusted. However, the host trusts device manufacturers and vendors to equip the secure world with TrustZone's secure boot protocol. This allows the host to establish trust in the secure world, which contains the policy-enforcement code. It is the host's responsibility to inspect the normal world memory snapshot to determine whether it is malicious, contains known exploitable vulnerabilities, or allows guests to bypass its memory modifications. From the guest device's perspective, the host may attempt to violate its security and privacy by accessing and modifying normal world memory. The guest relies on the vetting service, which it trusts, to determine the safety of the host's remote memory operations. Guests must keep their devices powered-on or use REM-suspend to ensure that verification tokens persist during their stay in the restricted space.

Out-of-Scope Threats. The guest device's normal world may contain zero-day vulnerabilities, such as a new buffer overflow in the kernel. The host may not be aware of this vulnerability, but a malicious guest may have a successful exploit that allows the host's policies to be bypassed. While such threats are out of scope, the host may require the guest's normal world to run a fortified software stack (*e.g.,* Samsung Knox [11] or MOCFI [29]) that implements defenses for common classes of attacks. The host could check this requirement during the inspection phase. A malicious guest device may also launch a denial-of-service attack, which will prevent the host from communicating with the secure world on the guest device. Such attacks can be readily detected by the host, which can prevent the device from checking-in. We also do not consider physical attacks whereby an adversarial guest attempts to bypass the host's memory updates by modifying the contents of the device's memory chip using external methods.

We restrict ourselves to guest devices that use the ARM TrustZone. It may still be possible for hosts to enforce usage policies on non-TrustZone devices using other means (see §4). However, it is not possible to provide strong security guarantees without trust rooted in hardware. While such "legacy" devices are still pervasive today, modern devices are outfitted with the TrustZone, and data from Samsung [11] indicates that millions of ARM TrustZone devices are already deployed. We hypothesize that in the future, hosts will have to contend with fewer legacy guest devices than they do today.

Finally, we only consider overt uses of guest devices in restricted spaces. Covert uses, where a guest stealthily smuggles a device into the restricted space without check-in and carefully avoids an electronic footprint (*e.g.,* by shielding the device from the host's WiFi access points), must still be addressed with traditional physical security methods.

2.3 Remote Memory Operations

We now discuss how hosts can use remote memory read and write operations to analyze and control guest devices. Our goal is to describe the power of remote memory operations as an analysis and control API. The discussion below is therefore intentionally broader than the features that we implemented in our prototype (a description of which appears in §2.6).

Analysis of Guest Devices. Hosts analyze memory snapshots of a guest's normal world kernel to determine its configuration and scan it for kernel-level malware (also called *rootkits*).

(**1**) **Retrieving configuration information.** When a guest device first checks-in, the host must determine the configuration of the device, so that it can suitably tailor further analysis of the device. The host determines the kernel version by inspecting code pages, thereby also allowing it to check if the guest has applied recommended security patches. To ensure that the normal world is free of malicious kernel code, the host compares a hash of each kernel code page against a whitelist, *e.g.,* of code pages in approved Android distributions [53, 83]. Additionally, the host must ensure that the kernel is configured to disallow well-known attack surfaces, *e.g.,* access to /dev/kmem and dynamic module loading. Finally, the host identifies addresses at which functions of a peripheral's driver are loaded, where they are hooked into the kernel and the addresses that store memory-mapped peripheral settings. To do so, it uses the recursive memory snapshot traversal technique described below. The host uses this information to design the set of memory updates that reconfigure the device to make it policy-compliant.

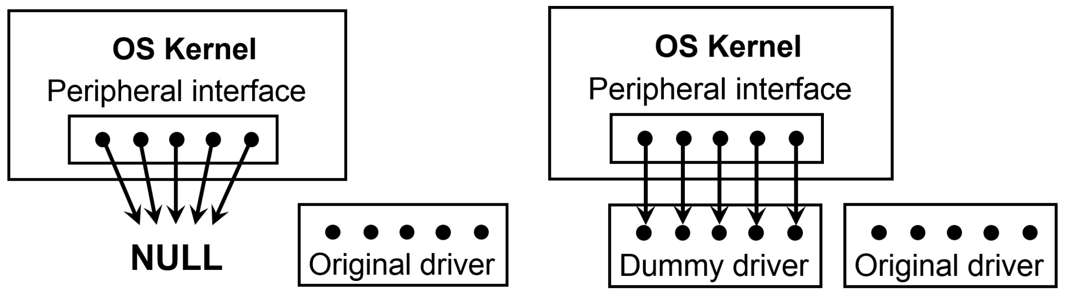
(**2**) **Detecting malicious data modifications.** Rootkits achieve malicious goals by modifying key kernel data structures [67, 13, 66]. The attack surface exposed by kernel data structures is vast. For instance, a rootkit could inject a device driver in kernel memory and modify kernel function pointers to invoke methods from this driver. Other examples of data structures that can be misused include process lists, entropy pools used by the kernel's random number generator, and access control structures [13, 66].

We now describe a generic approach, developed in prior work [67, 12, 21, 27, 41], that hosts can use to detect such malicious data modifications by analyzing the normal world's memory snapshot. The main idea is to recursively traverse the memory snapshot and reconstruct a view of the kernel's data structures, and use this view to reason about the integrity of kernel data. We assume that the host has access to the type declarations of the data structures used by the guest device's normal world kernel, *e.g.,* the sizes, layouts, and fields of every data structure. The host obtains this information from trusted repositories using the kernel version, extracted as discussed earlier.

Snapshot traversal starts from well-known entrypoints into the system's memory, *e.g.,* the addresses of the entities in System.map. When the traversal process encounters a pointer, it fetches the memory object referenced by the pointer and recurses until all objects have been fetched. This traversal process works even if the guest device uses address-space layout randomization (ASLR) techniques to protect its normal world, as is done on modern Android, iOS and Windows devices. Having reconstructed a view of kernel data structures, the host can then determine whether they have been maliciously modified. For example, it could check that function pointers in the kernel point to functions defined in the kernel's code space [67]. Similarly, the host can check that the kernel's data structures satisfy invariants that typically hold in an uncompromised kernel [12]. Prior research projects have explored in-depth the full power of memory snapshot analysis for rootkit detection [67, 12, 21, 27, 41]. We do not further elaborate on these rootkit detection policies because they are orthogonal to our focus. Our own prototype implementation only showcases a simple rootkit detection policy that ensures the integrity of the normal world's system call table. However, our design allows hosts to implement any of the complex rootkit detection policies described in prior work.

Analysis of memory snapshots is a powerful approach for hosts to obtain strong assurances about guest devices. A rootkit-infected OS kernel can be reliably diagnosed *only by externally observing its code and data*, *e.g.,* using memory snapshots as already discussed. Prior techniques that enforce policies on guest devices by using security-enhanced, policy-enforcing normal world kernels (*e.g.,* [40, 20, 58, 74, 65, 73, 78, 57, 16]) can also benefit from our approach to establish normal world kernel integrity to hosts. Although recent work [98] has explored cache-only normal-world rootkits on ARM TrustZone devices (which do not leave a memory footprint), the large majority of known rootkits operate by modifying kernel memory and can be detected via memory snapshot analysis.

We have restricted our discussion and our prototype implementation to analyzing the normal world's kernel memory snapshot. In theory, it is possible for a host to also request and analyze the normal world's user-space memory, *e.g.,* for malicious apps that reside in memory or on the file system. However, in practice, user-space memory may contain sensitive information stored in apps, which guests may be unwilling to share with hosts. For example, guests can configure their vetting service to mark as UNSAFE host requests to fetch user-space memory pages, as we do in our prototype (see §2.5).



(a) NULLifying the interface **(b) Installing a dummy driver**

Each device driver exposes an interface and is linked to the kernel via function pointers. Part (a) shows how to uninstall the peripheral by making the kernel's device interface point to NULL bytes. Part (b) shows how to uninstall the peripheral by unlinking the original driver and instead linking a dummy driver.

Figure 2.3: Uninstalling peripheral device drivers using remote write operations to kernel memory.

To ensure user-space security, hosts can leverage the normal world kernel after establishing that it is benign, *e.g.,* using the snapshot traversal methods described above. The host can require the normal world kernel to execute a mutually-agreed-upon anti-malware app in user-space. At check-in, the host scans the process list in the device's kernel memory snapshot to ensure that an anti-malware is executing. This app can check user-space memory and the file-system for malicious activity. At check-out, it can ensure that the same app is still executing by comparing its process identifier to the value obtained at check-in,[3] thereby ensuring that the anti-malware app was active for the duration of the guest's stay.

Control over Guest Device Peripherals. Hosts control the availability and configuration of peripherals on guest devices via remote memory updates to the devices. After analyzing the guest's memory snapshot, hosts prepare a set of memory updates to control various peripherals on guest devices. These updates are used to simply uninstall peripherals that may be misused violate the host's policies. Our overall approach to controlling peripherals is to update peripheral device drivers. On modern OSes, each peripheral has an interface within the kernel. This interface consists of a set of function pointers that are normally set to point to the corresponding functions within the peripheral's device driver, which communicates with the peripheral.

[3]The security of this scheme is based on the fact that PIDs on UNIX systems are, for all practical purposes, unique on a given system. For example, while they can be recycled, it requires a large counter to wrap around.

We adopted two broad strategies to update device drivers:

(1) **Nullifying interfaces (Figure 2.3(a)).** This approach simply sets the function pointers in the peripheral's interface to NULL. If the kernel checks these pointers prior to invoking the functions, it will simply return an error code to the application saying that the device is not installed. This approach has the advantage of only involving simple writes to the kernel (NULL bytes to certain addresses), which can easily be validated as safe if the guest so wishes. However, we found in our evaluation (§2.6) that this approach can crash the device if the kernel expects non-NULL pointers.

(2) **Dummy drivers (Figure 2.3(b)).** In this approach, the host writes a dummy driver for the peripheral and links it with the kernel in place of the original driver. If the dummy driver simply return a suitable error code rather than communicating with the peripheral, it has the effect of uninstalling the peripheral. The error code is usually bubbled up to and handled by user apps. Some apps may not be programmed to handle such errors, so an alternative approach could be for the dummy driver to return synthetic peripheral data instead of error codes [14]. Dummy drivers also offer fine-grained peripheral control. For example, with 3G/4G, it may be undesirable to simply uninstall the modem to disable voice messaging because it also prevents the guest from making emergency calls. The host can avoid this by designing a dummy driver that allows calls to emergency numbers alone, while disabling others. In this approach, the host introduces new driver code into the guest. From the guest's perspective, this code is untrusted and must be safety-checked by the vetting service.

For the above approaches to be effective, the guest must not have access to certain attack vectors that can be used to bypass the host's memory updates. The onus of ensuring these attack vectors are precluded resides with hosts, who must carefully design their policies to analyze guest device memory snapshots. For example, the analysis must ensure that the `/dev/kmem` interface is not available to guests, that dynamic module loading is disallowed, and that peripheral registers are not mapped into user-space memory using the `mmap` interface. Likewise, the snapshot analysis must carefully account for all the interfaces exported by peripheral device drivers and aliases to the functions implemented in them to ensure that an unlinked driver is no longer reachable from any paths in the normal world kernel.

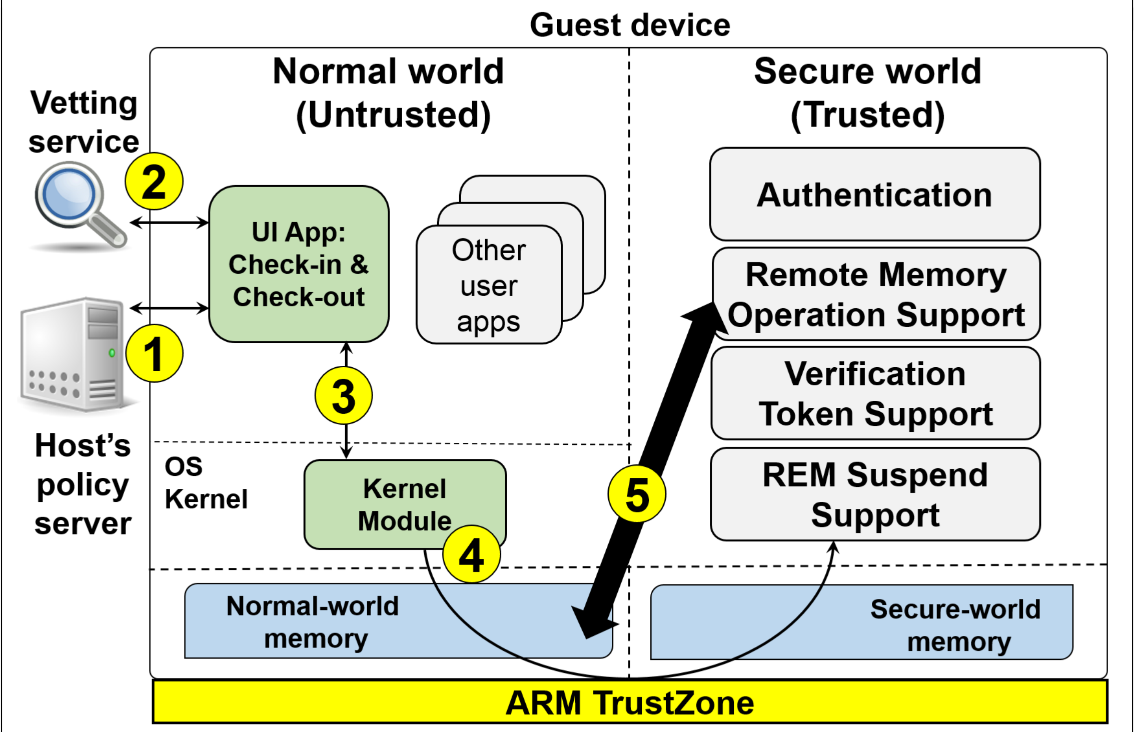
2.4 Policy Enforcement

We now present the design of our policy enforcement mechanism, which executes in the guest's secure world. The host must establish a channel to communicate with the guest's secure world. This channel must be integrity-protected from adversaries, including the guest's untrusted normal world. One way to set up such a channel is to configure the secure world to exclusively control a communications peripheral, say WiFi, and connect to the host without involving the normal world. Thus, the secure world must also execute the code necessary to support this peripheral. For peripherals such as WiFi, this would require several thousand lines of code from the networking stack to run in the secure work.

Our design aims to minimize the functionality that is implemented in the secure world. In our design, the normal world is assigned all peripherals on the guest device and therefore controls all external communication from the device. It establishes the communication channel between the secure world and the host. All messages transmitted on the channel are integrity-protected by the message sender using cryptographic checksums. The secure world itself provides support for just four key operations: mutual authentication (§2.4.1), remote memory operations (§2.4.2), verification tokens (§2.4.3), and REM-suspend (§2.4.4).

Guest devices are therefore set up as shown in Figure 2.4. Within the normal world, the end-user's interface is a user-level app (called the UI app) that allows him to interact with the host for device check-in and check-out. The app interacts with the components in the secure world via a kernel module. The host sends a request to perform remote memory operations on the guest device to the app. The app determines the safety of this request using the vetting service (§2.5), and forwards the request to the kernel module, which invokes `smc` to world switch into secure world. The components of the secure world then perform the request and communicate any return values to the host via the UI app. All messages include a message-authentication code computed using a key established during the mutual authentication step.

We do not place any restrictions on how the host and guest device communicate. Thus, the host's policy server could be hosted on the cloud and communicate with the guest device over WiFi or 3G/4G. Alternatively, the host could install physical scanners at a kiosk or on the entry-way to the restricted space. Guest devices would use Bluetooth, NFC, or USB to pair



① The host communicates with the UI app on the guest and sends requests to perform remote memory operations. ② The UI app uses the vetting service to determine the safety of the request. ③ If determined to be safe, the UI app forwards this request to the supporting kernel module. ④ The kernel module invokes the secure world by performing a world switch. ⑤ The secure world performs the requested memory operations on the normal world memory on behalf of the host. The components in the normal world (the UI app and kernel module) are untrusted. We rely on ARM TrustZone's secure boot to establish trust in the secure world.

Figure 2.4: Guest device setup showing components of the policy enforcement mechanism.

with the scanner and use it to communicate with the host.

The core mechanisms that run in the secure world of the guest device have two key features. They are *policy-agnostic* in that the same mechanisms can be used to enforce a variety of host policies. The narrow read/write interface is also *platform-agnostic*, and allows the same mechanisms to work irrespective of whether the normal world runs Android, iOS or Windows. This approach shifts complex device analysis and policy formulation tasks to the hosts. Hosts would naturally need to have separate modules to analyze and formulate memory updates for various normal world OSes.

Let host's public/private keypair be PubKeyH, PrivKeyH.
Let guest's public/private keypair be PubKeyG, PrivKeyG.

1. **Guest → Host**: PubKeyG, Certificate(PubKeyG)
2. **Host → Guest**: PubKeyH, Certificate(PubKeyH)
3. Guest and host verify Certificate(PubKeyH) and Certificate(PubKeyG)
4. **Host → Guest**: M, $\mathsf{Enc}_{\mathsf{PrivKeyH}}(M)$ (*i.e.,* host signs M),
   where M is $\mathsf{Enc}_{\mathsf{PubKeyG}}(k_s$, timestamp)
5. Guest verifies host's digital signature, decrypts M to obtain k_s, and checks timestamp

Figure 2.5: Mutual authentication and establishment of k_s.

2.4.1 Authentication

The host and guest device begin by mutually authenticating each other (Figure 2.5). We assume that both the host and the guest device have public/private key pairs with digital certificates issued by a certifying authority. The guest device stores its private key PrivKeyG in its secure world, thereby protecting it from the untrusted normal world.

Authentication is akin to SSL/TLS handshakes. The host and the guest exchange public keys and validate the certificates of these keys with the issuing authority. The host then computes a session key k_s, which is then transmitted to the client over a secure channel. Note that k_s is only used to protect the integrity of messages transmitted between the guest and the host and not their secrecy. The key k_s is stored in secure world memory, and is invisible to the normal world. It persists across REM-suspends of the guest device, but is erased from memory if the device is rebooted.

As with SSL/TLS, the ability of this protocol to resist man-in-the-middle attacks depends on the host and guest device's ability to validate each other's public keys. We assume that device vendors would provision PubKeyG for individual devices and register them with a certifying authority. Hosts register PubKeyH in much the same manner as is done for Web services today. While there is a case to be made that the certifying authority model has its limitations in the era of the Web and smart devices [22], we note that our use of authentication is entirely standard—to validate the host's and guest device's identities and to establish a session key k_s. Thus, we think that the other parts of our policy enforcement mechanism will work as-is with alternative authentication schemes, *e.g.,* those that use identity-based encryption.

2.4.2 Remote Reads and Writes

Remote Reads. During check-in the host typically requests the guest to send raw memory pages from the normal world for analysis. The UI app receives this request and performs a world switch to complete the request. The world switch suspends the UI app and transfers control to the secure world. Each request is a set (or range) of virtual memory addresses of pages that must be sent to the host. The host also includes a message-authentication code, a SHA1-based HMAC in our case, with the request. The HMAC is computed on the body of the request using the key k_s negotiated during authentication.

The secure world checks the integrity of the request using the HMAC. This step is necessary to ensure that the request was not maliciously modified by the untrusted components in the normal world. The secure world then translates each virtual page address in the request to a physical page address by consulting the page table in the normal world kernel. In this case, the page table will correspond to the suspended context in the normal world, *i.e.,* that of the UI app, into which the running kernel is also mapped. It then creates a local copy of the contents of this physical page from the normal world, and computes an HMAC over the page (again using k_s). The page and its HMAC are then copied to a buffer in the normal world, from where they can be transmitted to the host by the UI app. The host checks the HMAC and uses the page for analysis. This process could be iterative, with the host requesting more pages from the guest device based upon the results of the analysis of the memory pages received up to that point.

Both the host and the secure world are isolated from the normal world, which is untrusted. We only rely on the normal world kernel to facilitate communication between the host and the secure world. Moreover, both the host and the secure world use HMACs to protect the integrity of messages transmitted via the normal world. The normal world may drop messages and cause a denial-of-service attack; however, such attacks are outside our threat model (see §2.2.3). The host can therefore reliably obtain the memory pages of the normal world to enable the kinds of analyses described in §2.3. Communication between the host and the secure world is not confidential and is therefore not encrypted.[4] Thus, a malicious normal world kernel can potentially snoop on the requests from the host to fetch pages and attempt to remove the

[4]The host and guest could communicate over SSL/TLS, but this channel on the guest ends at the UI app, which runs in the normal world.

infection to avoid detection. However, this would have the desirable side-effect of cleaning the guest device at check-in.

Remote Writes. The host reconfigures the guest by modifying the running state of the normal world kernel via remote memory updates. The host sends the guest a set of triples $\langle$*vaddr*$_i$, *val*$_i$, *old-val*$_i\rangle$ together with an HMAC of this request. The normal world conveys this message to the secure world, which verifies its integrity using the HMAC. For each virtual address *vaddr*$_i$ (which refers to a memory location in the virtual address space of the UI app) in the request, the secure world ensures that the current value at the address matches *old-val*$_i$. If *all* the *old-val*$_i$ values match, the secure world replaces their values with *val*$_i$; else the *entire* operation is aborted.

Because the normal world is frozen during the course of this operation, the entire update is atomic with respect to the normal world. When a remote write operation succeeds, the secure world computes and returns a verification token to the host. If not, it returns an ABORT error code denoting failure.

The host's memory update request is aborted if the value stored at *vaddr*$_i$ does not match *old-val*$_i$. This design feature is required because the host's remote read and write operations do not happen as an atomic unit. The host remotely reads pages copied from the normal world's memory, analyzes them and creates remote write request using this analysis. During this time, the normal world kernel continues to execute, and may have updated the value at the address *vaddr*$_i$.

If the memory update is aborted, the host repeats the operation until it succeeds. That is, it refetches pages from the guest, analyzes them, and creates a fresh update. In theory, it is possible that the host's memory updates will abort *ad infinitum*. However, for the setting that we consider, aborts are rare in practice. This is because our write operations modify the addresses of peripheral device driver hooks. Operating systems typically do not change the values of device driver hooks after they have been initialized at system boot.

In theory, a remote memory write can also abort if the virtual address *vaddr*$_i$ referenced in the request is not mapped to a physical page in memory, *i.e.,* if the corresponding page has been swapped out to persistent storage. In practice, however, we restrict remote writes to kernel data pages that are resident in physical memory, as is the case with device drivers and pages that store data structures of peripherals. Thus, we do not observe Aborts due to a failure to resolve $vaddr_i$s.

It is possible to completely avoid such problems by designing the both the read and write operations to complete within a single world switch. During this time, the normal world remains frozen and cannot change the view of memory exported to the host. The read and write operations will therefore happen as an atomic unit from the normal world's perspective. However, in this case, the secure world must have the ability to directly communicate with the host. As previously discussed, we decided against this design because it has the unfortunate consequence of bloating the functionality to be implemented in the secure world. Thus, we make the practical design tradeoff of minimizing the functionality of the secure world while allowing the rare remote write failure to happen.

Note that our approach uses virtual addresses in remote memory operations. In doing so, we implicitly trust the integrity of page tables in the normal world kernel, which are used to translate these virtual addresses to physical ones. Recent work has demonstrated address-translation redirection (ATRA) attacks that work by maliciously modifying page table entries and the page table base register [46]. An ATRA attack effectively hides malicious code and data modifications by creating shadow pages containing unmodified code and data, and maliciously modifying page table entries to redirect requests from a security monitor to these shadow pages. On ARM TrustZone devices, it is possible to defend against such attacks by ensuring that all normal world page table updates are shepherded by the secure world. This is implemented by modifying the normal world kernel to invoke the secure world (via `smc`) for page table updates, and implementing a suitable security policy within the secure world to ensure the integrity of these updates [11, 37]. We have not implemented this defense in our prototype and doing so will require additional code in the secure world. However, we note that such a defense is already implemented as part of the secure world in Samsung Knox [11]. We hypothesize that a solution that integrates our approach with Knox will therefore be robust against ATRA attacks.

2.4.3 Verification Tokens

The host receives a verification token from the secure world upon successful completion of a remote write operation that updates normal world memory. A verification token $\mathsf{VTok}[r]$ is the following value: $r||MemState||\mathsf{HMAC}_{k_s}[r||MemState]$ where $MemState$ is $\langle vaddr_1,\ val_1\rangle||\ldots||\langle vaddr_n,\ val_n\rangle$, the set of $vaddr_i$ modified by the remote write, and the new values val_i at these locations. The token $\mathsf{VTok}[r]$ is parameterized by a random nonce r. This nonce can either be provided by the host together with the remote write request, or can be generated by the secure world.

Verification tokens allow the host to determine whether the guest attempted to revert the host's memory updates, either maliciously or by turning off the guest device. To do so, the host obtains a verification token $\mathsf{VTok}[r_{checkin}]$ upon completion of check-in, and stores this token for validation. During checkout, the host requests a validation token $\mathsf{VTok}[r_{checkout}]$ from the guest over the same virtual memory addresses. The secure world accesses each of these memory addresses and computes the verification token with $r_{checkout}$ as the nonce. The host can compare the verification tokens $\mathsf{VTok}[r_{checkin}]$ and $\mathsf{VTok}[r_{checkout}]$ to determine whether there were any changes to the values stored at these memory addresses.

The nonces $r_{checkin}$ and $r_{checkout}$ ensure the freshness of the tokens $\mathsf{VTok}[r_{checkin}]$ and $\mathsf{VTok}[r_{checkout}]$. The use of k_s to compute the HMAC in the verification token ensures that the token is only valid for a specific device and for the duration of the session, *i.e.,* until checkout or until the device is powered off, whichever comes earlier. Because k_s is only stored in secure world memory, it is ephemeral and unreadable to the normal world. Any attempts to undo the host's memory updates performed at check-in will thus be detected by the host.

2.4.4 Restricted Space Mode (REM) Suspend

If a guest device is rebooted, the host's updates to device memory are undone and k_s is erased from secure world memory, thereby ending the session. However, it is sometimes necessary to suspend the device in the restricted space, *e.g.,* to conserve battery power. We design REM-suspend to handle such cases and allow the session key k_s to persist when the device is woken.

The ARM TrustZone allows a device to be configured to route certain interrupts to the secure world [5]. We route and handle power-button presses and low-battery events in the secure world by prompting the user to specify whether to REM-suspend the device. When a guest device is checked into a restricted space, we configure the default power-down option to be REM-suspend; the default reverts to the traditional power-down sequence when the device checks out. The user can consciously choose to bypass REM-suspend, in which case the device shuts down the traditional way, thus ending the session. The same happens if the device shuts down due to other causes, *e.g.,* power loss caused by removing the device's battery.

When the guest device REM-suspends, the secure world checkpoints normal world memory, which contains the host's updates, and the key k_s, which are both restored when the device is woken up. The main challenge is to protect the confidentiality of k_s. The device user is untrusted, and can read the contents of persistent storage on the device; k_s must thus be stored encrypted with a key that is not available to the device user.

To achieve this goal, we leverage a feature referenced in the ARM TrustZone manual [5, §6.3.1], which provisions a device with a statistically-unique one-time programmable secret key that we will refer to as $\mathsf{K_{Dev}}$. $\mathsf{K_{Dev}}$ is located in an on-SoC cryptographic accelerator, and accessible only to secure world software [5, §6.3.1]. $\mathsf{K_{Dev}}$ cannot be read or changed outside the secure world, other bus masters or the JTAG [47]. $\mathsf{K_{Dev}}$ allows confidential data to be encrypted and bound to the device, and has previously been referenced in other research [80, 23, 71, 80]. Note that $\mathsf{K_{Dev}}$ is not the same as PrivKeyG, the device's private key.

In REM-suspend, the secure world first checkpoints normal world memory and CPU registers, and suspends the execution of the normal world. It sets a bit $\mathsf{B_{REM}}$ to record that the device is REM-suspended. It stores the checkpoint and $\mathsf{B_{REM}}$, together with an HMAC of these values under k_s on the device's persistent storage. It also stores to persistent storage the value of k_s encrypted under $\mathsf{K_{Dev}}$. The untrusted device user does not know $\mathsf{K_{Dev}}$, and therefore cannot forge the encrypted value of k_s or retrieve the cleartext value of k_s. The HMACs under k_s protect the integrity of the normal world checkpoint and $\mathsf{B_{REM}}$. An alternative way to protect the integrity of the normal world checkpoint and $\mathsf{B_{REM}}$ is to store them in the replay-protected memory block (RPMB), a trusted storage partition available on many mobile devices that come equipped with a embedded multi-media storage controller. The RPMB offers integrity-protection for stored data, ensures data freshness by protecting against replay attacks, and has been leveraged in

Component Name	LOC
Secure World (TCB)	
Memory manager	1,381
Authentication	1,285
Memory ops. & verif. tokens	305
REM-suspend	609
SHA1+HMAC	861
X509	877
RSA	2,307
Normal World	
Kernel module	93
UI app	72

Table 2.1: Sizes of components on the guest.

recent work [71]. However, even with this alternative the confidentiality of k_s needs to be protected by encrypting it using $\mathsf{K_{Dev}}$.

When the device is woken up, the secure world uses $\mathsf{B_{REM}}$ to check if the device is REM-suspended. If so, it uses $\mathsf{K_{Dev}}$ to retrieve k_s, verifies the integrity of the normal world checkpoint and $\mathsf{B_{REM}}$ using their HMACs, and starts the normal world from this checkpoint. The device resumes execution under the same session and continues to produce verification tokens if requested by the host.

The original ARM TrustZone manual [5] described $\mathsf{K_{Dev}}$ in the context of a hypothetical device, and $\mathsf{K_{Dev}}$ is not part of the core specification of the TrustZone architecture. As such, it is not clear how many deployed devices support $\mathsf{K_{Dev}}$; for example, it is not supported by the TrustZone-enabled board that we used for our prototype implementation. Many emerging ARM TrustZone-based security solutions [80, 23, 71] rely on the existence of $\mathsf{K_{Dev}}$, and it is likely that future revisions of the TrustZone architecture will incorporate such a key. The REM-suspend protocol can be used on any device that supports $\mathsf{K_{Dev}}$ or a cryptographic key with similar properties, *i.e.,* a hardware-provisioned key only accessible from the secure world. Note that guest devices that do not support such a key can still be restricted using our approach. However, the only shortcoming is that without REM-suspend, a power-down event will undo the memory updates requested by the host, and clear k_s, thereby terminating the session with the host.

Peripheral uninstalled	**Approach used (from Figure 2.3)**	**Device used**	**Bytes added or modified**	**Vetting time (sec.)**	**Verification token (bytes)**
USB (webcam)	NULLification	i.MX53	104	-	260
USB (webcam)	Dummy driver	i.MX53	168	2.22	388
Camera	NULLification	Nexus	140	-	332
Camera	Dummy driver	Nexus	224	2.19	500
WiFi	Dummy driver	Nexus	152	5.58	356
3G (Data)	Dummy driver	Nexus	192	2.15	436
3G (Voice)	Dummy driver	Nexus	124	2.15	300
Microphone	Dummy driver	Nexus	164	2.27	380
Bluetooth	Dummy driver	Nexus	32	2.52	116

Table 2.2: Peripherals uninstalled using remote write operations to a guest device.

2.5 Guest Privacy and Security

We built a vetting service trusted by guests to determine the safety of a host's request. We built it as a cloud-based server, to which the guest device forwards the host's memory updates together with a copy of its normal world memory image (via the UI app). We assume that the device and the vetting service have authenticated each other as in Figure 2.5 or use SSL/TLS to obtain a communication channel with end-to-end confidentiality and integrity guarantees. It may also be possible to implement vetting within the secure world itself. However, we chose not to do so to avoid bloating the secure world.

The vetting server checks the host's requests against its safety policies and returns a SAFE or UNSAFE response to the device. The response is bound with a random nonce and an HMAC to the original request in the standard way to prevent replay attacks. The secure world performs the operations only if the response is SAFE. Guests can configure the vetting server with domain-specific policies to determine safety. Our prototype vetting service, which we built as a plugin to the Hex-Rays IDA toolkit [1], analyzes memory images and checks for the following safety policies. Although simple and based on conservative whitelisting, in our experiments, the policies could prove safety without raising false positives.

- **Read-safety.** For each request to read from address $vaddr_i$, we return SAFE only if $vaddr_i$ falls in a pre-determined range of virtual addresses. In our prototype, acceptable address ranges only include pages that contain kernel code and kernel data structures. The vetting server returns UNSAFE if the read request attempts to fetch any addresses from kernel buffers that store user app data, or virtual address ranges that lie in app user-space memory.

• **Write-safety.** Our prototype currently only allows write requests to NULLify peripheral interfaces or install dummy drivers that disable peripherals. We use the following safety policy for dummy drivers. For each function f implemented in the dummy driver, consider its counterpart f_{orig} from the original driver, which the vetting service obtains from the device's memory image. We return SAFE only if the function f is identical to f_{orig}, or f's body consists of a single return statement that returns a *valid* error code (*e.g.,* -ENOMEM). We define an error code as being valid for f if and only if the same error code is returned along at least one path in f_{orig}. The intuition behind this safety check is that f does not modify the memory state of the device or introduce new and possibly buggy code, but returns an error code that is acceptable to the kernel and client user apps. For more complex dummy drivers that introduce new code, the vetting service could employ a traditional malware detector or more complex program analyses to scan this code for safety.

We implemented the above safety policies in a 190-line Python plugin to the IDA toolkit. In the following section, we report the performance of the vetting server as it established the safety of various host requests to uninstall guest device peripherals. Although we have only explored the simple safety policies discussed above, the vetting service can implement more complex policies, and we plan to experiment with such policies in future work. For example, although our read-safety policy ensures that only kernel code and data pages can be sent to the host, even these pages may compromise the guest's privacy. The buffer cache and various buffers used by the networking stack reside in kernel data pages, and may store sensitive user information. A more nuanced read-safety policy would identify which memory addresses store such data and mark as UNSAFE any host requests to fetch data from those addresses. Note that implementing more complex vetting policies will increase the code-base of the vetting service, which the guest trusts. However, this complexity does not affect the size of the TCB running on the guest device.

2.6 Implementation and Evaluation

We implemented our policy enforcement mechanism on an i.MX53 Quick Start Board from Freescale as our guest device. This board is TrustZone-enabled and has a 1GHz ARM Cortex A8 processor with 1GB DDR3 RAM. We chose this board as the guest device because it offers open, programmable access to the secure world. In contrast, the vendors of most commercially-available TrustZone-enabled devices today lock down the secure world and prevent any modifications to it. A small part of main memory is reserved for exclusive use by the secure world. On our i.MX53 board, we assigned the secure world 256MB of memory, although it may be possible to reduce this with future optimizations. The normal world runs Android 2.3.4 atop Linux kernel version 2.6.35.3.

We built a bare-metal runtime environment for the secure world, just enough to support the components shown in Figure 2.4. This environment has a memory manager, and a handler to parse and process commands received from the host via the normal world. To implement cryptographic operations, we used components from an off-the-shelf library called the ARM mbed TLS (v1.3.9) [4]. Excluding the cryptography library, our secure world consists of about 3,500 lines of C code, including about 250 lines of inline assembly. The secure world implements all the features described in §2.4, except for one minor deviation in the implementation of the REM-suspend protocol. The i.MX53 does not support $\mathsf{K}_{\mathsf{Dev}}$, so our prototype implements REM-suspend assuming that such a key is available and can be fetched from hardware.

Table 2.1 shows the sizes of various components. We used mbed TLS's implementation of SHA1 and HMACs, RSA and X509 certificates. As shown in Table 2.1, the files implementing these components alone comprise only about 4,000 lines of code. In addition to these secure world components, we built the kernel module and the UI app (written as a native daemon) for the normal world, comprising 165 lines of code. We implemented a host policy server that authenticates guest devices, and performs remote memory operations. We conducted experiments to showcase the utility of remote reads and writes to enforce the host's policies on the guest. The guest and the host communicate over WiFi.

Guest Device Analysis. To illustrate the power of remote memory read operations to perform device analysis, we wrote a simple rootkit that infects the guest's normal world kernel by hooking its system call table. In particular, it replaces the entry for the close system call to instead point to a malicious function injected into the kernel. The malicious functionality ensures that if the process invoking close calls it with a magic number, then the process is elevated to root. Although simple in its operation, Petroni and Hicks [67] show that over 95% of all rootkits that modify kernel data operate this way.

We were able to detect this rootkit on the host by remotely reading and analyzing the guest's memory pages. We remotely read pages containing the init, text and data sections of kernel memory. Our analyzer, a 48 line Python script, reads the addresses in the system call table from memory, and compares these entries with addresses in System.map. If the address is not included, *e.g.,* as happens if the entry for the close system call is modified, it raises an error. For more sophisticated rootkits that modify arbitrary kernel data structures, the host can use complex detection algorithms [67, 12, 21] based on the recursive snapshot traversal method outlined in §2.3.

For the above experiment, it took the secure world 54 seconds to create an HMAC over the memory pages that were sent to the host (9.2MB in total). It takes under a second to copy data from the normal world to the secure world and vice versa. It may be possible to accelerate the performance of the HMAC implementation using floating point registers and hardware acceleration, but we have not done so in our prototype.

USB	***MobileWebCam***	***Camera ZOOM FX***	***Retrica***	***Candy Camera***	***HD Camera Ultra***
Passive	AppErrMsg	AppErrMsg	AndroidErrMsg	AppErrMsg	AndroidErrMsg
Active	AppErrMsg	AppErrMsg	AppErrMsg	AppErrMsg	AppErrMsg
Camera	***Camera for Android***	***Camera MX***	***Camera ZOOM FX***	***HD Camera for Android***	***HD Camera Ultra***
Passive	AndroidErrMsg	AppErrMsg	AppErrMsg	AndroidErrMsg	AndroidErrMsg
Active	BlankScreen	AppErrMsg	AndroidErrMsg	BlankScreen	BlankScreen
WiFi	***Spotify***	***Play Store***	***YouTube***	***Chrome Browser***	***Facebook***
Passive	LostConn	LostConn	LostConn	LostConn	LostConn
Active	LostConn	LostConn	LostConn	LostConn	LostConn
3G (Data)	***Spotify***	***Play Store***	***YouTube***	***Chrome Browser***	***Facebook***
Passive	LostConn	LostConn	LostConn	LostConn	LostConn
Active	LostConn	LostConn	LostConn	LostConn	LostConn
3G (Voice)	***Default call application***				
Passive	AppErrMsg: Unable to place a call				
Active	AppErrMsg: Unable to place a call				
Microphone	***Audio Recorder***	***Easy Voice Recorder***	***Smart Voice Recorder***	***Sound and Voice Recorder***	***Voice Recorder***
Passive	AppErrMsg	AppErrMsg	AppErrMsg	AppErrMsg	AppErrMsg
Active	EmptyFile	EmptyFile	EmptyFile	EmptyFile	EmptyFile

We use *Passive* to denote experiments in which the user app was not running when the peripheral's driver was replaced with a dummy, and the app was started after this replacement. We use *Active* to denote experiments in which the peripheral's driver was replaced with a dummy even as the client app was executing. ① AppErrMsg denotes the situation where the user app starts normally, but an error message box is displayed within the app after it starts up; ② BlankScreen denotes a situation where the user app displayed a blank screen; ③ LostConn denotes a situation where the user app loses network connection; ④ EmptyFile denotes a situation where no error message is displayed, but the sound file that is created is empty; ⑤ AndroidErrMsg denotes the situation where the user app fails to start (in the passive setting) or a running app crashes (in the active setting), and the Android runtime system displays an error.

Table 2.3: Results of robustness experiments for user apps.

Guest Device Control. We evaluated the host's ability to dynamically reconfigure a guest device via remote memory write operations. For this experiment, we attempted to disable a number of peripherals from the guest device. However, the i.MX53 board only supports a bare-minimum number of peripherals. As proof-of-concept, we therefore tested the effectiveness of remote writes on a Samsung Galaxy Nexus smart phone with a Texas Instruments OMAP 4460 chipset. This chipset has a 1.2GHz dual-core ARM Cortex-A9 processor with 1GB of RAM, and runs Android 4.3 atop Linux kernel version 3.0.72. This device has a rich set of peripherals, but its chipset comes with TrustZone locked down, *i.e.,* the secure world is not accessible to third-party programmers. We therefore performed remote writes by modifying memory using a kernel module in its (normal world) OS. Thus, while remote writes to this device do not enjoy the security properties described in §2.4, they allow us to evaluate the ability to uninstall a variety of peripherals from a running guest device.

Table 2.2 shows the set of peripherals that we uninstalled, the method used to uninstall the peripheral (from §2.3), the device on which we performed the operation (i.MX53 or Nexus), and the size of the write operation, *i.e.,* the number of bytes that we had to modify/introduce in the kernel. We were able to uninstall the USB on the i.MX53 and the camera on the phone by NULLifying the peripheral interface. For other peripherals, we introduced dummy drivers designed according to the safety criterion from §2.5. We also used dummy drivers for the USB and the camera to compare the size of the write operations. In this case, the size of the write includes both the bytes modified in the peripheral interface and the dummy driver functions. For the 3G interface, we considered two cases: that of disabling only 3G data and that of only disabling calls. Our experiment shows it is possible to uninstall peripherals without crashing the OS by just modifying a few hundred bytes of memory on the running device.

For each uninstalled peripheral, Table 2.2 shows the time taken by the vetting service to determine the safety of the write operation (using the policy from §2.5). Our vetting service runs on a quad-core Intel i5-4960 CPU running at 3.5Gz, with 16GB of memory. Table 2.2 also shows the size of the verification token generated by the secure world for the write operation. The size of the verification token grows linearly with the size of the write operation, but is just a few hundred bytes in all cases. On the i.MX53, it took the secure world under 6 milliseconds to generate the verification tokens. This shows that it is practical for the host to request the guest device to resend the verification token at periodic intervals during its stay in the restricted space. Installing a dummy driver disables the peripheral, but how does it affect the user app that is using the peripheral? To answer this question, we conducted two sets of experiments involving a number of client user apps that leverage the peripherals shown in Table 2.2. In the first set of experiments, which we call the *passive setting*, we start with a configuration where the client app is not executing, replace the device driver of the peripheral with a dummy, and then start the app. In the second set of experiments, called the *active setting*, we replace the peripheral's device driver with the dummy as the client app that uses the peripheral is executing.

Table 2.3 shows the results of our experiments. For both the passive and active settings, we observe that in most cases, the user app displays a suitable error message or changes its behavior by displaying a blank screen or creating an empty audio file. In some cases, particularly in the passive setting, the app fails to start when the driver is replaced, and the Android runtime displays an error that it is unable to start the app.

2.7 Summary

In this chapter, we develop mechanisms that allow hosts to analyze and regulate ARM TrustZone-based guest devices using remote memory operations. These mechanisms can be implemented with only a small amount of trusted code running on guest devices. The use of the TrustZone allows our approach to provide strong guarantees of guest policy-compliance to hosts. Our vetting service allows guests to identify conflicts between their privacy goals and the hosts' usage policies.

While this chapter demonstrates technical feasibility of our approach, questions about its adoptability in real-world settings remain to be answered. For example, we can imagine our solution to be readily applicable in settings such as federal or corporate offices and examination halls, where restricted spaces are clearly demarcated and the expectations on guest device usage are clearly outlined. Will it be equally palatable in less stringent settings, such as social gatherings, malls or restaurants? A meaningful answer to this question will require a study of issues such as user-perception and willingness to allow their devices to be remotely analyzed and controlled by hosts. We hope to investigate these and other issues in follow-on research.

Chapter 3

Regulating Smart Devices with SEAndroid

In this chapter, we present a policy enforcement scheme to provide a higher-level abstraction for smart devices in restricted spaces as an alternative approach to the previous work. In our approach, we leverage Security-Enhanced Linux in Android (SEAndroid) for fine-grained access control, and use Near field communication (NFC) for secure communications.

3.1 Introduction

The number and a diversity of personal computing devices, which include smartphones, tablets, and watches, have rapidly increased over the last few years. Furthermore, having evolved annually, these devices have incorporated a faster CPU, larger memory capacity, and a variety of sensors. Smart devices with the developments bring convenience to our daily lives; nevertheless, we face difficult challenges that the smart devices may be misused at the places we visit.

For instance, these days, the Bring Your Own Device (BYOD) trend is continuously growing because it has many advantages such as cost-savings, reducing security concerns and enhancing productivity. As stated in the report [35], some aspect of BYOD will be supported by 90% of organizations, and employees will use two times as many employee-owned devices for work as enterprise-owned devices by 2018. However, sensitive information in work environments can be leaked by leveraging the sensors of smart devices from malicious end-users or apps.

Smart devices might be used to take pictures, or record videos in less strict environments such as movie theaters, gym locker rooms and restaurants. There are many actual cases of invasion of privacy and illegal activities. For example, a person could face jail since she posted a nude photo of a woman in the locker room of a fitness center on social media [42]. In movie theaters, it is easy to illegally record movies using the camera on smart devices, whereas it is difficult to detect that. According to the report [15], film piracy results in loss of the U.S. economy $20.5 billion every year, and it is also estimated that over 90% of illegal Internet content is derived from undetected recordings in movie theaters [69].

Therefore, we need a method to control smart device use in such restricted spaces. In this chapter, we introduce an approach to dynamically enforce policies on smart devices in restricted spaces. Our goal is to provide a universal and practical solution for fine-grained access control of smart devices. In an enterprise environment, the mobile device management (MDM) solutions have been introduced for BYOD. However, guests would not want to install their MDM solutions on the guests' devices for one-time use. Our approach can be a separate mechanism of current MDM solutions. And the existing context-based access control system may not be accurate enough to differentiate between locations [75]. To overcome its limitations, we utilize the NFC on smart devices as nearby location-based service. Consequently, we can ensure where guests are in if the NFC is not compromised.

Our prior research [19] worked on low-level mechanisms for restricted spaces. Hence, we built a small trusted computing base (TCB) that can detect rootkits and also regulate smart devices in the restricted spaces. Nevertheless, there are weaknesses in our work due to the system design. First, guests might consider that the previous work is too intrusive. In the system, a guest sends the kernel memory image on the guest's device to a host, so that guests' privacy problems could be concerned. Second, the previous work was designed at low-level, so the system could be difficult to not only be deployed to smart devices, but also be maintained and configured as a vendor's perspective. Third, the policy language in the previous work is not user-friendly because the system reads and writes kernel memory at low-level, which is suitable for peripherals access control, but app or file level access control on smart devices. To overcome these shortcomings, we accordingly need a higher-level abstraction mechanism for fine-grained access control as an alternative version to our previous work.

After Security Enhanced for Android (known as SEAndroid) was introduced in 2013, the Android platform with SEAndroid has been distributed over 88% in the Android market [70] nowadays. Likewise, recently most Android devices have adopted SEAndroid. In this chapter, we propose a secure policy enforcement system called ForceDroid that can dynamically enforce

SEAndroid policies on smart devices for fine-grained access control. Hosts and guests securely communicate with each other to exchange policy information and compliances by using their NFC-enabled devices. Guest devices have the predefined policies corresponding to hosts' policies in their restricted spaces. The predefined SEAndroid policies on guest devices are enforced during secure connections. Our prototype also leverages the ARM TrustZone architecture [5] as the root of trust.

To summarize our main contributions.

- *No privacy concerns.* We solve the privacy concerns of *remote memory operations* in our previous work. In this section, to do so, our approach relies on SEAndroid as a higher-level abstraction mechanism instead of remote memory operations at low-level.
- *Enforcement mechanism.* We present the design of a mechanism for hosts to enforce predefined policies on guest device use in restricted spaces. We provide a fine-grained access control mechanism by leveraging SEAndroid, which means that the system design allows us to more easily configure and maintain smart devices policies.
- *Prototype implementation.* We demonstrate a prototype implementation of the mechanism using an NFC controller for practical use, and the ARM TrustZone hardware as a root of trust on guest devices.

3.2 Background

We briefly provide some background on SE for Android, NFC, ARM TrustZone, and OP-TEE. We then describe an overview of the restricted space model.

3.2.1 SEAndroid

For hardening Android security, Security Enhancements for Android (SE for Android) [82, 81] was introduced to provide mandatory access control (MAC) over all processes using Security-Enhanced Linux (SELinux) [85]. Security-Enhanced Linux (SELinux), as part of the Linux Security Module (LSM), enforces MAC over traditional discretionary access control (DAC).

Unlike DAC, MAC restricts access to objects (e.g., file, socket) based on the ability of a subject (e.g., process). The SEAndroid policy rules have the following form.

allow domains types:classes permissions;

A domain is a subject label, and a type is an object label. The policy rule defines which domain of subjects have permissions to access which types and classes of objects. For instance, the rule,

allow forcedroid forcedroid_data_file:file { *rw_file_perms* }*;*

allows the processes in the *forcedroid* domain to open, read, and write the files of *forcedroid_data_file* type. In order to avoid writing policies mistakenly, SEAndroid also has *neverallow* rules that should never be allowed. In ForceDroid, we leverage SEAndroid to provide fine-grained access control for peripherals, apps and file system.

3.2.2 NFC

Near field communication (NFC) is a set of short-range wireless technologies to establish communication between two electronic devices, or an NFC tag and an electronic device within a distance of 4cm [62]. The NFC technology is useful for various applications such as contactless payment systems, keycards and sharing information.

NFC devices provide three modes of operation: reader/writer mode, peer-to-peer (P2P) mode, card emulation mode. First, the reader/writer mode allows the NFC device to read or write information on passive NFC tags. Second, the P2P mode allows two NFC devices to communicate with each other for exchanging data. Third, the card emulation mode allows the NFC device to perform as an NFC card, which enables users to accomplish payment transaction [61].

NFC is vulnerable to numerous kinds of attacks such as eavesdropping and Man-in-the-Middle (MITM) attack due to its lack of secure communication channel. Therefore, we take NFC security standards (NFC-SEC) to safely exchange a shared secret key for further communications. The NFC-SEC uses the Elliptic Curves Diffie-Hellman (ECDH) protocol for key agreement and the AES algorithm for data encryption and integrity [2]. In our design, ForceDroid runs on the host card emulation (HCE) mode for guests and reader/writer mode for hosts to securely communicate between hosts and guests.

3.2.3 OP-TEE

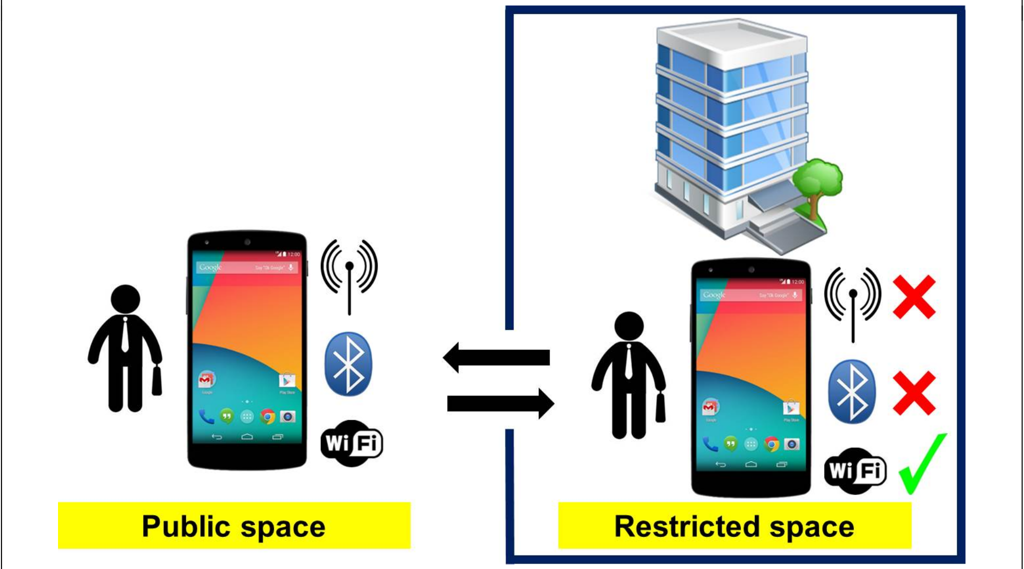
OP-TEE [64] is an Open-source Portable Trusted Execution Environment for ARM TrustZone-enabled devices. The OP-TEE includes the Trusted OS running in the secure world, the secure monitor for switching between two worlds, the Linux kernel Trusted Execution Environment (TEE) driver that helps to communicate between normal world user space and secure world, and the client API's running in the normal world user space. The OP-TEE, as an isolated execution environment, runs beside a rich OS such as Android and Linux to provide secure computing to normal world applications in the rich OS. In ForceDroid, we employ OP-TEE as a secure world OS for trusted computing.

3.2.4 Restricted Space Model

We present an overview of the restricted space model. In this section, we do not limit to the workspace as the restricted space, which can be not only the workspaces, but also public restricted spaces such as movie theaters, airports and saunas. As depicted in Figure 3.1, when a guest accesses to a restricted space, the guest checks-in each of the guest's devices during entry. While check-in, the guest devices follow the ForceDroid check-in procedure. After checked-in, the guest devices can be used under the policies of the host in the restricted space. When the guest exits the restricted space, the guest devices do the check-out procedure. Then ForceDroid restores the original policies after the verification procedure.

3.3 Threat Model

We summarize our threat model. We assume that the guest and the host do not trust each other. However, the host trusts only the secure world of the guest's device since the host trusts the TrustZone with its secure boot protocol from device manufacturers and vendors. In our threat model, all the peripherals of the device except NFC can be tampered in the normal world because the secure kernel manages NFC peripheral in the secure world. We assume that the existing SEAndroid policies in the secure storage on guest devices do not have vulnerabilities. The normal world of the guest device may contain zero-day vulnerabilities such as a newly-discovered buffer overflow in the kernel. In this case, a malicious guest may have an exploit



A guest checks-in devices when entering a restricted space. During check-in, a host enforces their policies on the guest device. In this example, the host allows only the use of WiFi on the device. When the guest checks-out the device, the host requests to restore the original policies the guest device.

Figure 3.1: An overview of our restricted space model.

to bypass enforcing the host's policies, and the host may not be aware of these vulnerabilities. Such threats are outside the scope of our work, but the host may protect itself by requiring the guest's work environment to operate a hardened software stack (e.g., Samsung Knox [11, 78] or MOCFI [29]).

It is certainly possible for a guest to bypass the host's policies by not declaring a device during check-in and using it covertly within the restricted space. As long as the guest carefully configures the device to avoid accessing any of the host's resources in the restricted space, e.g., its WiFi access points, and remain stealthy, the device cannot be detected by the host. Our focus in this paper is to address policy enforcement for overt uses of smart devices. Covert uses, such as the above, are out of the scope of the mechanism developed in this paper. Instead, we assume that traditional methods such as physical security checks are necessary to detect covertly-hidden devices.

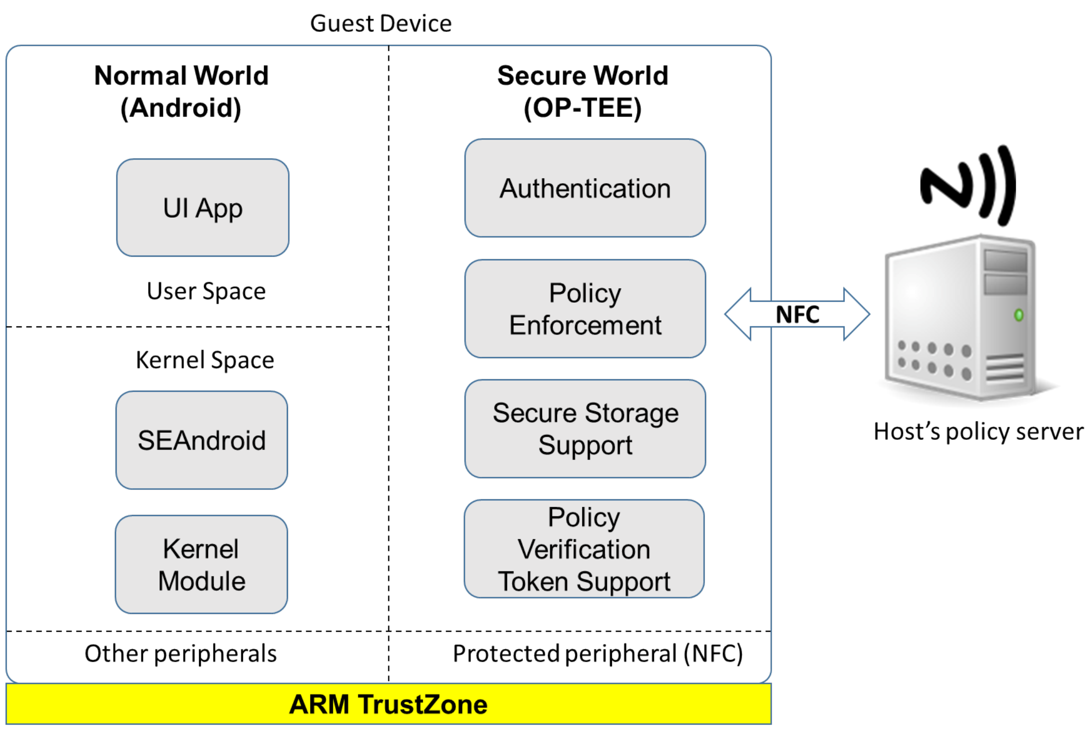


Figure 3.2: ForceDroid Architecture

3.4 ForceDroid Architecture

In this section, we describe the design of the ForceDroid system. An overview of the Force-Droid design is shown in Figure 3.2.

3.4.1 Design Overview

Our design aims to provide secure policy enforcement for Android. ForceDroid consists of two parts: host's policy server and ForceDroid on guest device. The host runs a policy server that communicates with guest devices in its restricted space. The normal world of each guest executes the end-user's work environment, and can run a full-fledged mobile operating system in our prototype the normal world executes Android. Because this code is under the control of the end-user, the normal world is untrusted. The secure world of the guest runs a TCB that accepts and processes the operations remotely-initiated by the host to inspect the guest device and regulate the use of its peripherals. For this setup to work, we need a communication channel that allows the host to securely relay its requests to the guest and obtain the guest's responses.

The host has its owns EC public/private keypair: PubKeyH, PrivKeyH
The guest has its own EC public/private keypair: PubKeyG, PrivKeyG

1. **Guest** → **Host**: PubKeyG || Nonce$_G$
2. **Host** → **Guest**: PubKeyH || Nonce$_H$
3. **Guest**: computes k_s=KDF(Nonce$_G$, Nonce$_H$, ID$_G$, ID$_H$)
   computes HMAC$_G$=f(k_s, ID$_G$, ID$_H$, PubKeyG, PubKeyH)
4. **Guest** → **Host**: HMAC$_G$
5. **Host**: computes k_s=KDF(Nonce$_G$, Nonce$_H$, ID$_G$, ID$_H$)
   check HMAC$_G$ and computes HMAC$_H$=f(k_s, ID$_G$, ID$_H$, PubKeyG, PubKeyH)
6. **Host** → **Guest**: HMAC$_H$
7. **Guest**: check HMAC$_H$

The host and the guest set $k_s$for secure communications.

Figure 3.3: Authentication and key establishment

In particular, the channel must not allow an attacker, such as the untrusted code executing in the normal world, to tamper with messages transmitted on it.

One way to build such a channel is to set up the secure world to directly communicate with the host. In this case, the secure world would exclusively manage a communication peripheral, such as NFC, and establish a connection with the host without involving the normal world. Therefore, the code to support this peripheral must also execute within the secure world. With NFC, for instance, this would need several thousand lines within the TCB. We successfully ported an NFC device driver to the TCB.

ForceDroid on guest devices relies on ARM TrustZone for trusted computing, and ForceDroid on the policy server verifies policy compliances on the guest devices. In the normal world, a user-level client app (called the UI app) is the end-users' interface to allow end-users to perform check-in and check-out procedure. The secure world is an arbitrator between the host and the normal world of the guest device. The components in the secure world perform operations such as reloading policies via a kernel module.

As we described earlier, we can therefore overcome the shortcomings of our previous work [19]. However, we can still take advantages of some parts of the previous work such as guest device analysis and REM-suspend. For instance, the guest device analysis feature can protect SEAndroid in ForceDroid.

ForceDroid on Policy Server. A ForceDroid Policy Server, which can be an NFC enabled computing device, consists of three modules: authentication, policy enforcement, and policy verification. The authentication module is in charge of authenticating the guest devices during NFC communications. The policy server can share a temporary secret key with the guest devices after being successfully authenticated. The policy enforcement module securely sends the host's policy information according to the host's requirements since all communications between the policy server and the guest device are protected by the secret key. The policy verification module acquires a verification token from the component of the secure world after successful completion of policy enforcement operations that reloads a new policy file and validates the verification token.

ForceDroid on Guest Device. ForceDroid on guest devices provides four key operations: authentication, policy enforcement, secure storage, and verification token support.

- **Authentication** The authentication module is to securely communicate with a host. This module obtains a shared key with a host by the NFC-SEC protocol [2].
- **Policy Enforcement** The policy enforcement module is responsible for reloading a predefined policies file in the secure storage through the kernel module. The module also verifies the requested policies from the host.
- **Secure Storage Support** The secure storage support module stores secret keys in the secure storage so that the keys are protected from compromising attackers. The secure storage also keeps predefined SEAndroid policy files that are corresponding to various restricted spaces settings.
- **Policy Verification Token Support** The policy verification token support module computes a verification token when the policy is correctly loaded in SEAndroid. The module then sends the verification token to the host.

3.4.2 ForceDroid Workflow

Figure 3.4 shows how the ForceDroid check-in protocol works. First of all, a guest device sends a new session request to a host. The host and the secure world of the guest device share a secret key from the authentication and key agreement protocol based on the NFC-SEC

The host and the guest share a secret key, $k_s$via the NFC-SEC protocol.
1. **Guest** $\rightarrow$ **Host**: Requesting a new session
2. **Host** $\rightarrow$ **Guest**: PolicyID || Nonce_H || HMAC_{k_s}(PolicyID || Nonce_H)
3. **Guest**: Invoke loading the policies
4. **Guest** $\rightarrow$ **Host**: Nonce_G || HMAC_{k_s}(Policy || Nonce_G)
5. **Host**: Verify HMAC_{k_s}(Policy || Nonce_G)

Figure 3.4: ForceDroid Protocol: Check-in

protocol. The host then sends its policy information to the secure world of the guest. The policy enforcement module on the guest device loads the requested policies from the secure storage. The policy verification module sends the policy information and the cryptographic hash value of policy compliance to the host. The host verifies the policy compliance token and sends an acknowledgment back to the secure world. After successful policy verification, the verification token is stored in the secure storage for check-out. The check-out protocol runs similar to the check-in protocol.

3.4.3 Authentication and Key Agreement

This step enables the guest and the host to identify each other mutually. We first assume that the host and the guest have their Elliptic Curve (EC) public/private key pairs, and the guest device protects its private key in its secure storage so that the untrusted normal world cannot access the private key. In our work, we adopted the NFC-SEC protocol [2].

Figure 3.3 shows the authentication and key agreement protocol of ForceDroid. First of all, the host and the secure world of the guest device exchange their public keys with nonces. The secure world generates a session key k_s computed by a key derivation function (KDF) and a message authentication code, HMAC_G using the guest device's private key and the host's public key. The message is then transferred to the host over a secure channel. The host receives the message from the guest device and also computes the session key k_s using its private key and the guest's public key. After verifying HMAC_G, the host computes HMAC_H and send it to the guest. Lastly, the guest validates HMAC_H received from the host. The session key k_s is stored in the secure storage of the guest device, thereby protecting the key from the normal world. The guest device's public/private key pair and the session key persists in the secure storage even if the device is rebooted. During checking-out, the session key is discarded.

3.4.4 Policy Enforcement

ForceDroid utilizes SEAndroid for fine-grained access control. The policy enforcement module in the secure world validates a host's policy request. If the request is valid, at the request of the host, the policy enforcement module reads the encrypted policy file corresponding to the host's request from the secure storage. The predefined policy files are encrypted by the secret key of ForceDroid. After decrypting the policy file, the policy enforcement module runs the kernel module to load the policies in SEAndroid. To load SEAndroid policies, we bring the *selinux_-android_reload_policy* function in the libselinux library of Android systems to our approach. In SEAndroid, there is a pseudo-file system called SELinuxFS that provides the interface between the kernel and userspace for access control. The policy enforcement module remotely writes SEAndroid policies to the kernel via the SELinuxFS (e.g., */sys/fs/selinux/load*).

3.4.5 Verification Token

Upon successful completion of policy enforcement, the verification token support module generates a verification token and send it to the host. A verification token *VTok[n]* is the value *n||Policy||HMACks[n||Policy]*, with a random nonce *n*. Verification tokens enable the host to determine whether the guest attempted to manipulate the policies on the guest devices, by either malicious actions or by powering off the guest device. The host acquires a verification token $VTok[n_{checkin}]$ after completion of check-in, and keep this token for validation in the secure storage. While checkout, the host requests a validation token $VTok[n_{checkout}]$ from the guest device over the current policies. The module read the current policy file and generates the verification token with $n_{checkout}$. The host can check the verification tokens $VTok[n_{checkin}]$ with $VTok[n_{checkout}]$ to determine whether there were any modifications to the SEAndroid policies. The verification token is only valid for a specific device until check-out, so that the host will detect any attempts to undo the policy changes performed at check-in.

Component Name	LOC
Guest Device	
UI app	186
Kernel module	113
NFC device driver	3,723
Host	
Policy server app	181

Table 3.1: Sizes of ForceDroid prototype components

3.5 Implementation and Evaluation

We implemented our ForceDroid prototype on an i.MX6Q development board as a guest device. This board is an ARM TrustZone-enabled single board computer that has a 1GHz ARM Cortex A9 processor and 1GB DDR3 RAM. The normal world runs Android 6.0.1 atop Linux kernel version 3.14. We ported OP-TEE 2.4 [64] as a trusted OS to the i.MX6Q board for the secure world, and the OP-TEE provides memory manager and cryptographic operations for trusted computing.

To securely communicate between the host and the guest, we used an NXP OM5578/PN7150ARD NFC controller kit [3], so we ported an NXP NFC driver to the secure world on the i.MX6Q board. The NFC device driver mainly handles an Inter-Integrated Circuit (I^2C) interface and a general purpose input/output (GPIO) interface. The P2P mode of NFC is not appropriate for bi-direction communications. Thus, the guest device runs on the NFC host-based card emulation mode, and the host device works on the NFC reader/writer mode.

We used the LibTomCrypt [51] library to perform cryptographic functions for the prototype. The library provides support for HMAC, AES encryption, RSA encryption, and signing. We also implemented an NFC application on Nexus 5 as a host policy server, and the host policy server can be any NFC-enabled computer devices.

Table 3.1 shows the sizes of ForceDroid prototype components. The client application and the kernel module for OP-TEE comprise about 300 lines of C code. The NFC device driver in the secure world has around 4,000 lines of C code. The host policy server application, which communicates via NFC and performs cryptographic operations, consists of approximately 200 lines of Kotlin code.

```
# Allow system to talk to usb device
allow system_server usb_device:chr_file rw_file_perms;
allow system_server usb_device:dir r_dir_perms;
```

Figure 3.5: SEAndroid policy example

```
/dev/tee0 u:object_r:tznfc_device:s0
/dev/teepriv0 u:object_r:tznfc_device:s0
/dev/tee1 u:object_r:tznfc_device:s0
/dev/teepriv1 u:object_r:tznfc_device:s0
/system/bin/tee-supplicant u:object_r:tznfc_exec:s0
/system/bin/tee_helloworld u:object_r:tznfc_exec:s0
/data/tee(/.*)? u:object_r:tznfc_data_file:s0
```

Figure 3.6: ForceDroid file contexts

We performed experiments to show the utility of reloading host's policies on the guest. We tested the effectiveness of enforcing policies on a USB-based camera (Logitech C260) for disabling the USB peripheral with the i.MX6Q board, and the camera of Nexus 5 for disabling its functions. ForceDroid can easily enforce any other policies on recent SEAndroid-based Android phones. We utilize SEAndroid to provide fine-grained access control, and it runs in enforcing mode.

Figure 3.5 shows part of SEAndroid policy example for the i.MX6Q board. If these rules are commented out, then the *system_server* domain cannot access the USB peripheral. Our approach can provide various policies depending on restricted spaces with SEAndroid that supports peripheral, app, and file system access control.

Customization SEAndroid Policy. We declare ForceDroid domain type and define rules in Figure 3.7. For instance, the policy states ForceDroid can access system_data_files and ForceDroid files. We also need to define file contexts for the ForceDroid domain. Figure 3.6 shows ForceDroid file contexts.

The rule, *allow init kernel:security load_policy*, has a vulnerability to assist privilege escalation. The *init* process does not need *load_policy* permission from the kernel because the *init* process loads a default policy before SEAndroid has initialized. We therefore define a separate domain, and prohibit *load_policy* permission from other domains access except for the ForceDroid domain that communicates with only the secure world components.

Performance Evaluation. We evaluated the performance regarding reloading policies and

```
type tznfc, domain;
type tznfc_exec, exec_type, file_type;
type tznfc_data_file, file_type, data_file_type;

init_daemon_domain(tznfc)

allow tznfc system_data_file:dir write;
allow tznfc system_data_file:dir add_name;
allow tznfc system_data_file:dir create;
allow tznfc tznfc_data_file:dir create_dir_perms;
allow tznfc tznfc_data_file:file rename setattr getattr link cre-
ate rw_file_perms create_file_perms ;
allow tznfc system_data_file:file rename setattr getattr link
create rw_file_perms create_file_perms ;
allow tznfc self:capability dac_override;
allow tznfc tznfc_device:chr_file rx_file_perms rw_file_-
perms ;
allow tznfc kernel:security load_policy ;
allow tznfc selinuxfs:file open write getattr ;
```

Figure 3.7: ForceDroid domain and rules

communicating through NFC between guests and hosts. When a guest and a host have a shared session key, the check-in or check-out procedure to reload policy rules via an NFC tap takes 438 ms on average.

Security Analysis. Rooted or malicious applications might be able to directly access sensitive data of other applications. Thus, our system allows only ForceDroid domain processes to reload predefined policies by the trusted application in the secure world. Hosts can send only requests with policy information to enforce policies on guest devices, so that the guest devices are protected from the hosts and do not have privacy concerns.

3.6 Use cases

The host can configure peripherals on the guest device by reloading a policy file in SEAndroid. We consider various peripherals and describe several scenarios where control over them would be useful.

3.6.1 Peripherals

- ***Camera*** The use of the camera is perhaps the most obvious way for guests to violate privacy and confidentiality (e.g., [91, 92]). In the federal and enterprise settings, employees may photograph or videotape sensitive documents and meetings, while in social settings, the camera can be used to record conversations. In such settings, the camera can be disabled at check-in.

- ***Microphone*** Much like the camera, the microphone can also be used to violate privacy and confidentiality. However, disabling the microphone on devices such as smartphones may not be acceptable to guests because it also prevents them from having phone conversations. It may be possible for the host to configure the microphone's driver or the application level so that the microphone is activated only when a call is placed, and is deactivated otherwise. However, this facility can possibly be misused by malicious guests to record meetings by simply placing a call during the meeting. Therefore, it may be desirable to simply disable the microphone except when the guest places an outgoing call to an emergency number, and require guests to check-out if they wish to place other phone calls.

- ***Networking*** Enterprises (and federal institutions) today often disallow employees and visitors from connecting their personal devices to the corporate network. This is typical to avoid exfiltrating information outside the enterprise. To prevent exfiltration, the enterprise can disable the use of 3G/4G, and restrict the device to only WiFi. Because the enterprise controls the WiFi network, it can therefore regulate what data is accessible to the device and the external hosts to which the device can connect. In an examination setting, for example, the proctor/university can similarly restrict networking interfaces when students check-in their devices.

- ***Detachable Storage.*** USB dongles and flash drives are extensively used to copy files across devices. However, they have also served as the vehicle for malware infections (e.g., the Conficker worm [84]) and can be used by malicious guests to exfiltrate sensitive data from the host. Many enterprises only have informal guidelines discouraging their employees from using dongles and flash drives to copy files. With our approach, the host can simply disable the drivers for USB and flash drives at check-in, thereby preventing the use of detachable storage media within the restricted space.

- ***Bluetooth*** Smart glasses and smart watches rely on Bluetooth to pair with a more powerful hub, such as a smart phone, via which they connect to the external world. While it is generally possible to control their connection to the outside world by constraining the networking interfaces on the hub device, in some scenarios it may be necessary to prevent the device from pairing with the hub. For example, a student wearing a prescription smart glass may pair the device with his smart phone and use it to access class notes stored on the phone. Disabling Bluetooth prevents this channel, but allows the student to use the glass for vision correction.

3.6.2 Scenarios

- ***Exam*** In the exam settings, proctors' smart devices can be hosts. Students establish secure connections with the proctors' devices by an NFC tap. During secure communications, students agree to enforce the required exam policy on their smart devices. Proctors can also determine whether the students' devices are policy-compliant on proctors' devices. In this scenario, our system plays an essential role to ensure that the exam policy properly regulates the student's devices use in the classroom. After exams, students' devices can automatically revert to their original policies through the check-out procedure.

- ***Car*** The growth of smartphones uses lead to distracted driving. According to the report from the U.S Department of Transportation, in 2015, an approximated 30,000 people were injured in car accidents involving cell phone activities while driving [60]. In this scenario, a car can be a host and the smart device of a driver can be a guest device. Disabling unnecessary peripherals or apps could protect drivers from distracted driving.

- ***Movie theater*** Recording films is illegal in movie theaters, and it is challenging to catch illegal acts by using smart devices. Our approach can disallow the camera on visitors' smartphones in the movie theaters. This mechanism could combine with digital ticketing service in the near future. When visitors leave movie theaters, their original policies will be restored on their devices.

3.7 Summary

The increasing ubiquity and capability of smart devices has driven society to build restricted spaces. We presented a mechanism for hosts to enforce host's policies on guest devices in restricted spaces. We show that the mechanism achieves this goal by loading predefined policies on guest devices. This approach provides an effective way for hosts to configure guest devices in accordance with their policies.

While technically feasible, our approach must overcome certain hurdles before it can be practically adopted. The foremost among these is end-user willingness to subject their devices to regulation in restricted spaces. It is possible that many users would just choose not to use their devices within the restricted space rather than give the host control over their devices (assuming they do not resort to using the devices covertly). However, given our increasing reliance on smart devices and the ways in which they are becoming integral parts of our daily lives, it is unclear going forward whether such simple "opt-out" solutions would even be a possibility. For example, opting-out would not be a practical solution for smart devices integrated with health monitoring and assistive functionality.

Chapter 4

Related Work

TrustZone Support. A number of projects have used TrustZone to build novel security applications. TrustDump [88] is a TrustZone-based mechanism to reliably acquire memory pages from the normal world of a device (Android LiME [39] and similar acquisition tools [90, 38, 87] do so too, but without the security offered by TrustZone). While similar in spirit to remote reads, TrustDump's focus is to be an alternative to virtualized memory introspection solutions for malware detection. Unlike our work, TrustDump is not concerned with restricted spaces, authenticating the host, or remotely configuring guest devices.

Samsung Knox [11] and SPROBES [37] leverage TrustZone to protect the normal world in real-time from kernel-level rootkits. These projects harden the normal world kernel by making it perform a world switch when it attempts to perform certain sensitive operations to kernel data. A reference monitor in the secure world checks these operations, thereby preventing rootkits. In our work, remote reads allow the host to detect infected devices, but we do not attempt to provide real-time protection from malware. Our work can also leverage Knox to enhance the security of the normal world (§2.2.3).

While we have leveraged TrustZone's ability to isolate secure world memory from the normal world, TrustZone supports additional features that can be used to explore alternative designs. One such feature is TrustZone's support for peripheral reassignment between the normal and secure worlds. Using this feature, a host could require all of a guest device's restricted peripherals to the trusted secure world during check-in. The secure world implements the equivalent of dummy drivers that control and therefore restrict peripherals.

The above design is a viable alternative that we plan to explore in future work. In our current design, we chose to explore the benefits and limits of remote memory operations because it allowed us to satisfy our goal of minimizing the size of the TCB on guest devices. The alternative design described above would require additional driver code to execute in the secure world. That said, even this design alternative can leverage some of the ideas from our current work. For example, peripherals assigned to the secure world at check-in can be reassigned to the normal world via a device reboot. A protocol based on REM-suspend can be used to save peripheral assignment state upon power-down events, and restore the peripheral assignment upon power-up in guest devices that support $\mathsf{K}_{\mathsf{Dev}}$. TrustZone has also been used to improve the security of user applications. Microsoft's TLR [80] and Nokia's ObC [48] use TrustZone to provide a secure execution environment for user apps, even in the presence of a compromised kernel. Other applications include ensuring trustworthy sensor readings from peripherals [54], securing mobile payments (*e.g.,* Apple Pay and Samsung Pay), mobile data billing [72], attesting mobile advertisements [50], and implementing the TPM-2.0 specification in firmware [71], protecting the peripherals of drones [55], mobile peripheral control [49], and protecting user's interactions and secret [97].

Enterprise Security. With the growing "bring your own device" (BYOD) trend, a number of research projects and enterprise MDM products (*e.g.,* [78, 57, 16]) have developed security solutions that enable multiple persona(*e.g.,* [40, 20, 8]) or enforce mandatory access control policies on smart devices (*e.g.,* [95, 85, 20, 40]). Prior work has also explored context-based access control and techniques for restricted space objects to push usage policies onto guest devices (*e.g.,* [63, 25, 58, 74, 65, 73]).

These projects tend to use one of two techniques. One is to require guest devices to run a software stack enhanced with a policy enforcement mechanism. For instance, ASM [40] introduces a set of security hooks in Android, which consult a security policy (installed as an app) that can be used to create multiple persona on a device. Each persona is customized with a view of apps and peripherals that it can use. Another approach is to require virtualized guest devices [8, 26, 6, 28]. In this approach, a trusted hypervisor on the guest device enforces isolation between virtual machines implementing different persona.

The main benefit of these techniques over our work is the greater app-level control that they provide. For example, they can be used to selectively block sensitive audio and blur faces by directly applying policies to the corresponding user apps [74, 44]. These techniques are able to do so because they have a level of semantic visibility into app-level behavior that is difficult to achieve at the level of raw memory operations. On the other hand, our approach enjoys two main benefits over prior work. First, our approach simplifies the design of the trusted policy-enforcing code that runs on guest devices to a TCB of just a few thousand lines of code. In contrast, security-enhanced OSes and virtualized solutions required hundreds of thousands of lines of trusted policy-enforcement code to execute on guest devices. Prior research has investigated ways to reduce the TCB, *e.g.,* by creating small hypervisors [86]. However, the extent to which such work on small hypervisors applies to smart devices is unclear, given that any such hypervisor must support a variety of different virtualization modes, guest quirks, and hardware features on a diverse set of personal devices. Santos et al. [24, 79] present a system that enables Android devices to disable specific functions dynamically for limited amounts of time.

The second benefit of our approach is that it provides security guarantees that are rooted in trusted hardware. Prior projects have generally trusted guest devices to correctly implement the host's policies. This trust can easily be violated by a guest running a maliciously-modified OS or hypervisor. It is also not possible for a host to obtain guarantees that the policy was enforced for the duration of the guest's stay in the restricted space. We leverage the TrustZone to offer such guarantees using verification tokens and REM-suspend.

Other Hardware Interfaces. Hardware interfaces for remote memory operations were originally investigated for the server world to perform remote DMA as a means to bypass the performance overheads of the TCP/IP stack [43, 9]. This work has since been repurposed to perform kernel malware detection [68] and remote repair [17]. These systems use a PCI-based co-processor on guests via which the host can remotely transfer and modify memory pages on the guest.

On personal devices, the closest equivalent to such a hardware interface is the IEEE 1394 (Firewire), which is available on some laptops. However, it is not currently available on smaller form-factor devices. Another possibility is to use the JTAG interface [47], which allows read/write access to memory and CPU registers via a few dedicated pins on the chipset. However, the JTAG is primarily used for debugging and is not easily accessible on consumer devices. One drawback of repurposing these hardware interfaces is that they cannot authenticate the credentials of the host that initiates the memory operation. Moreover, to use these hardware interfaces on guest devices, the host needs physical access to plug into them. Thus, these interfaces are best used when the guest can physically authenticate the host and trust it to be benign.

App Security. It is now well-known that many popular apps exfiltrate sensitive user data from smart devices [31]. Moreover, a significant fraction of apps (on Android) are over-privileged [10, 32] and end-users are poor at understanding the meaning of app permissions [34, 52]. Such apps can leverage the increasing array of sensors on modern smart devices in novel and dangerous ways [74, 89]. These threats will amplify in the future as we see an increasing number of augmented reality apps that continuously monitor sensor feeds and extract data from the device's environment. Some projects have attempted to rectify the situation by offering improved app permission models [33] or modifying the execution environment on the device to return "fake" sensor data to apps [14]. However, such techniques are usually ineffective when the device itself is compromised (e.g., via kernel rootkits), or if the user unintentionally installs a malicious app. Researchers have also investigated defenses tailored toward improving privacy in the presence of augmented reality apps [44, 45]. Our work can complement these efforts by giving hosts the ability to control peripherals below the app layer.

Hardening Smart Devices. Finally, the research community has addressed techniques to harden the software stack of smart devices. Samsung Knox [11], as previously discussed, provides the ability to detect certain classes of kernel-level rootkits in real time. MOCFI [29] enhances the mobile operating system by enforcing control-flow integrity properties, thereby mitigating the effect of attacks such as buffer overflow-based exploits. Airbag [96] employs lightweight virtualization to isolate user apps and prevent them from infecting the device's firmware or leaking sensitive information. At the app level, RetroSkeleton [30] rewrites Android apps to improve their security on commodity devices. These techniques can help improve the resilience of smart devices to attack. Our work allows hosts to remotely analyze smart devices via remote memory operations and verify that they are free of malware infection. To harden Android security, SEAndroid has also been researched for years. EASEAndroid [94] analyzes and refines SEAndroid policies based on audit logs from real-world devices using machine learning techniques. SPOKE [93] is a tool that collects domain knowledge from functional tests and analyzes the attack surface of SEAndroid policy rules.

Chapter 5
Conclusion

The objective of this dissertation is how to regulate smart devices in restricted devices. We have shown the notion of restricted spaces. As argued in the paper, the increasing ubiquity and capability of smart devices have driven society to create such restricted spaces. To promote the regulated use of smart devices in such spaces, we need systematic ways to enforce host-defined policies on guest devices. We see this dissertation as contributing two main conceptual advances.

The first is the design of a mechanism for hosts to remotely inspect and control guest devices. We showed that this could be achieved via a narrow interface that permits two simple operations, remote memory reads and writes. These operations provide an effective way for hosts to analyze and configure guest devices in accordance with their policies. We also showed that with hardware support from the ARM TrustZone, we can bootstrap the security of remote memory operations. The second conceptual contribution of this dissertation is the design of a mechanism to enforce hosts' policies on guest devices by leveraging SEAndroid to overcome the disadvantages of the first design. In the design, we also use NFC technologies for secure communications.

We now describe future directions in this area. Recently, wearable technology has been developed rapidly, so the popularity of wearable devices is increasing in the world. Wearable devices are also smart devices which can be worn on body or accessories such as smartwatches. Nowadays people bring not only smartphones but also smartwatches in their daily lives. Consequently, we also need a way to regulate wearable devices in restricted spaces. Our approaches can be directly applied to wearable devices. Nevertheless, we need a way to enforce policies on a primary device to spread required policies to all other smart devices at once because people will carry numerous smart devices in the near future.

As previously discussed, this paper illustrates the technical feasibilities of our approaches, but questions about their adoptabilities in the real world still remain to be answered. For instance, we believe that our solutions can be applied in restricted spaces such as enterprise setting. Will the solutions be applicable in less stringent environments, such as movie theaters or restaurants? User-perception and willingness are the key issues for policy enforcement in public restricted spaces. We hope to study the issues in future work.

References

[1] Hex-rays software: About IDA. `https://www.hex-rays.com/products/ida/index.shtml`.

[2] NFC-SEC Cryptography Standard using ECDH and AES. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-386.pdf`.

[3] PN7150 NFC Controller SBC Kit User Manual. `https://www.nxp.com/docs/en/user-guide/UM10935.pdf`.

[4] SSL Library ARM mbed TLS/PolarSSL. `https://tls.mbed.org`.

[5] ARM security technology – Building a secure system using TrustZone technology, 2009. ARM Technical Whitepaper. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[6] VMware news release — Verizon Wireless and VMware securely mix the professional and personal mobile experience with dual persona Android devices, October 2011. `http://www.vmware.com/company/news/releases/vmw-vmworld-emea-verizon-joint-10-19-11.html`.

[7] N. Anderson and V. Strauss. Cheating concerns force delay in SAT scores for South Koreans and Chinese. In *Washington Post*, October 30, 2014.

[8] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *ACM Symposium on Operating Systems Principles*, 2011.

[9] The InfiniBand Trade Association. The InfiniBand$^{\text{TM}}$ architecture specification. `http://www.infinibandta.org`.

[10] K. Au, B. Zhou, J. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *ACM Conference on Computer and Communications Security*, 2012.

[11] A. 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 *ACM Conference on Computer and Communications Security*, 2014.

[12] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5), 2011.

[13] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security & Privacy*, 2007.

[14] A. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *ACM HotMobile*, 2010.

[15] C. Bialik. Putting a Price Tag on Film Piracy. In *The Wall Street Jounal*, April 5, 2013. `https://blogs.wsj.com/numbers/putting-a-price-tag-on-film-piracy-1228`.

[16] Blackberry. Enterprise mobility management – Devices, Apps, Content. Productivity Protected. `http://us.blackberry.com/enterprise/solutions/emm.html`.

[17] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using backdoors. In *International Conference on Autonomic Computing*, 2004.

[18] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A. R. Sadeghi. Regulating smart personal devices in restricted spaces, July 2015. Rutgers University Computer Science Technical Report.

[19] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A. R. Sadeghi. Regulating arm trustzone devices in restricted spaces. In *ACM MobiSys*, 2016.

[20] S. Bugiel, S. Hauser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security Symposium*, 2013.

[21] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *ACM Conference on Computer and Communications Security*, 2009.

[22] J. Clark and P. C. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security & Privacy*, 2013.

[23] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[24] M. Costa, N. Duarte, N. Santos, and P. Ferreira. TrUbi: A System for Dynamically Constraining Mobile Devices within Restrictive Usage Scenarios. In *ACM Symposium on Mobile Ad Hoc Networking and Computing*, 2017.

[25] M.J. Covington, P. Fogla, Z. Zhan, and M. Ahamad. A context-aware security architecture for emerging applications. In *Annual Computer Security Applications Conference*, 2002.

[26] L. P. Cox and P. M. Chen. Pocket hypervisors: Opportunities and challenges. In *ACM HotMobile*, 2007.

[27] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprings with a practical memory analysis system. In *USENIX Security Symposium*, 2012.

[28] C. Dall and J. Nieh. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[29] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A-R. Sadeghi. MoCFI: Mitigating control-flow attacks on smartphones. In *Network & Distributed Systems Security Symposium*, 2012.

[30] B. Davis and H. Chen. Retroskeleton: Retrofitting Android apps. In *ACM MobiSys*, 2013.

[31] W. Enck, P. Gilbert, B-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[32] A. Felt, E. Chin, K. Greenwood, and D. Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security*, 2011.

[33] A. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *USENIX Workshop on Hot Topics in Security*, 2012.

[34] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *ACM Symposium on Usable Privacy and Security*, 2012.

[35] Gartner Says Tablets Are the Sweet Spot of BYOD Programs. `http://www.gartner.com/newsroom/id/2909217`.

[36] Gartner Says Worldwide Smartphone Sales to Slow in 2016. `http://www.gartner.com/newsroom/id/3339019`.

[37] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the TrustZone architecture. In *IEEE Mobile Security Technologies Workshop*, 2014.

[38] Google. Using DDMS for debugging. `http://developer.android.com/tools/debugging/ddms.html`.

[39] A. P. Heriyanto. Procedures and tools for acquisition and analysis of volatile memory on Android smartphones. In *11th Australian Digital Forensics Conference*, 2013.

[40] S. Heuser, A. Nadkarni, W. Enck, and A. R. Sadeghi. ASM: A programmable interface for extending Android security. In *USENIX Security Symposium*, 2014.

[41] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[42] P. Holley. A Playboy Playmate photographed a senior citizen naked at the gym. Now she's facing jail time. In *Washington Post*, November 5, 2016.

[43] Mellanox Technologies Inc. Introduction to InfiniBand, September 2014. `http://www.mellanox.com/blog/2014/09/introduction-to-infiniband`.

[44] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security Symposium*, 2013.

[45] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *IEEE Symposium on Security & Privacy*, 2013.

[46] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. Kang. ATRA: Address translation redirection attack against hardware-based external monitors. In *ACM Conference on Computer and Communications Security*, 2014.

[47] Joint Test Action Group (JTAG). 1149.1-2013 - IEEE Standard for test access port and boundary-scan architecture, 2013. `http://standards.ieee.org/findstds/standard/1149.1-2013.html`.

[48] K. Kostiainen, J. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *ACM Symposium on Information, Computer and Communications Security*, 2009.

[49] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. SeCloak: ARM Trustzone-based Mobile Peripheral Control. In *ACM MobiSys*, 2018.

[50] W. Li, H. Li, H. Chen, and Y. Xia. Adattester: Secure online advertisement attestation on mobile devices using trustzone. In *ACM MobiSys*, 2015.

[51] Libtomcrypt. `https://www.libtom.net/LibTomCrypt/`.

[52] J. Lin, S. Amini, J. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2012.

[53] L. Litty, A. Lagar-Cavilla, and D. Lie. Hypervisor support to detect covertly executing binaries. In *USENIX Security Symposium*, 2008.

[54] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *ACM MobiSys*, 2012.

[55] R. Liu and Mani Srivastava. PROTC: PROTeCting Drone's Peripherals through ARM TrustZone. In *ACM Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*, 2017.

[56] M. McGee. New website tracks "Glasshole-Free Zones," businesses that have banned Google Glass. In *Glass Alamanac (*`http://glassalmanac.com`*)*, March 2014.

[57] Microsoft. Microsoft Intune: Simplify management of apps and devices. `https://www.microsoft.com/en-us/server-cloud/products/microsoft-intune/`.

[58] M. Miettinen, S. Heuser, W. Kronz, A.-R. Sadeghi, and N. Asokan. ConXsense – Context profiling and classification for context-aware access control. In *ACM Symposium on Information, Computer and Communications Security*, 2014.

[59] A. Migicovsky, Z. Durumeric, J. Ringenberg, and J. Alex Halderman. Outsmarting proctors with smartwatches: A case study on wearable computing security. In *International Conference on Financial Cryptography and Data Security*, 2014.

[60] National Center for Statistics and Analysis. Distracted driving 2015. (Traffic Safety Facts Research Note. Report No. DOT HS 812 381). Washington, DC: National Highway Traffic Safety Administration, March 2017.

[61] Near-field communication. `https://en.wikipedia.org/wiki/Near_field_communication`.

[62] Near field communication Overview. `https://developer.android.com/guide/topics/connectivity/nfc/index.html`.

[63] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically-rich application-centric security in Android. In *Annual Computer Security Applications Conference*, 2009.

[64] OP-TEE. `https://op-tee.org`.

[65] S. Patel, J. Summet, and K. Truong. Blindspot: Creating capture-resistant spaces. In *Protecting Privacy in Video Surveillance*, 2009.

[66] N. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security Symposium*, 2006.

[67] N. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security*, 2007.

[68] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: A coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[69] Pirateeye. `http://www.pirateeye.com/piracy/film-entertainment/`.

[70] Distribution dashboard. `https://developer.android.com/about/dashboards/index.html`.

[71] H. Raj, S. Saroiu, A. Wolman, R. Aigner, P. England J. Cox, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. fTPM: A firmware-based TPM 2.0 implementation, November 2015. Microsoft Research Technical Report.

[72] H. Raj, S. Saroiu, A. Wolman, and J. Padhye. Splitting the bill for mobile data with simlets. In *ACM HotMobile*, 2013.

[73] N. Raval, A. Srivastava, K. Lebeck, L. P. Cox, and A. Machanavajjhala. MarkIt: Privacy markers for protecting visual secrets. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing UPSIDE Workshop*, 2014.

[74] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, and H. J. Wang. World-driven access control for continuous sensing. In *ACM Conference on Computer and Communications Security*, 2014.

[75] Bilal S., Oyindamola O., and Elisa B. Context-based access control systems for mobile devices. *IEEE Transactions on Dependable and Secure Computing*, 12(2), 2015.

[76] A.-R. Sadeghi, C. Stuble, and M. Winandy. Property-based tpm virtualization. In *Information Security Conference*, 2008.

[77] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, 2004.

[78] Samsung. KNOX workspace supported MDMs. `https://www.samsungknox.com/en/products/knox-workspace/technical/knox-mdm-feature-list`.

[79] N. Santos, N. Duarte, M. Costa, and P. Ferreira. A Case for Enforcing App-Specific Constraints to Mobile Devices by Using Trust Leases. In *USENIX Workshop on Hot Topics in Operating Systems*, 2015.

[80] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a Trusted Language Runtime for mobile applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[81] Security-Enhanced Linux in Android. `https://source.android.com/security/selinux`.

[82] Security Enhancements (SE) for Android. `http://seandroid.bitbucket.org`.

[83] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles*, 2007.

[84] S. Shin, G. Gu, N. Reddy, and C. Lee. A large-scale empirical study of Conficker. *IEEE Transactions on Information Forensics and Security*, 7(2):676–690, 2012.

[85] S. Smalley and R. Craig. Security enhanced Android: Bringing flexible MAC to Android. In *Network & Distributed Systems Security Symposium*, 2013.

[86] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *European Symposium on Computer Systems*, 2010.

[87] A. Stevenson. Boot into recovery mode for rooted and un-rooted Android devices. `http://androidflagship.com/605-enter-recovery-mode-rooted-un-rooted-android`.

[88] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. TrustDump: Reliable memory acquisition on smartphones. In *European Symposium on Research in Computer Security*, 2014.

[89] M. M. Swift, M. Annamalai, B. N. Bershad, and H. N. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), 2006.

[90] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from Android smartphones. *Digital Investigation*, 8(3-4), 2012.

[91] R. Templeman, M. Korayem, D. Crandall, and A. Kapadia. PlaceAvoider: Steering first-person cameras away from sensitive spaces. In *Network & Distributed Systems Security Symposium*, 2014.

[92] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *Network & Distributed Systems Security Symposium*, 2013.

[93] R. Wang, A. Azab, W. Enck, N. Li, P. Ning, X. Chen, W. Shen, and Y. Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *ACM Symposium on Information, Computer and Communications Security*, 2017.

[94] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *USENIX Security Symposium*, 2015.

[95] X. Wang, K. Sun, Y. Wang, and J. Jing. DeepDroid: Dyanmically enforcing enterprise policy on Android device. In *Network & Distributed Systems Security Symposium*, 2015.

[96] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. AirBag: Boosting smartphone resistance to malware infection. In *Network & Distributed Systems Security Symposium*, 2014.

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

[98] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou. Cachekit: Evading memory introspection using cache incoherence. In *IEEE European Symposium on Security and Privacy*, 2016.

Acknowledgment of Previous Publications

The text of this dissertation, in part or in full, is a reprint of the materials as it appears in [18, 19]. Chapter 2 is joint work with Ferdinand Brasser, Christopher Liebchen, and Ahmad-Reza Sadeghi.

[18] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A. R. Sadeghi. Regulating smart personal devices in restricted spaces, July 2015. Rutgers University Computer Science Technical Report

[19] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A. R. Sadeghi. Regulating arm trustzone devices in restricted spaces. In *ACM MobiSys*, 2016