

# Regulating the Use of Smart Personal Devices in Restricted Spaces

Ferdinand Brasser  
Technische Universität Darmstadt  
ferdinand.brasser@trust.cased.de

Daeyoung Kim  
Rutgers University  
daeyoung.kim@cs.rutgers.edu

Christopher Liebchen  
Technische Universität Darmstadt  
christopher.liebchen@trust.cased.de

Vinod Ganapathy  
Rutgers University  
vinodg@cs.rutgers.edu

Liviu Iftode  
Rutgers University  
iftode@cs.rutgers.edu

Ahmad-Reza Sadeghi  
Technische Universität Darmstadt  
ahmad.sadeghi@trust.cased.de

## ABSTRACT

Smart personal devices have become ubiquitous, and we increasingly rely on their use in our daily lives. Simultaneously, we as a society are also beginning to place restrictions on how these devices can be used in various environments. Examples of such *restricted spaces* abound in today's society. For example, enterprises and federal offices that store sensitive information place restrictions on the use of cameras and microphones in smart devices. In the classroom, students are often disallowed from using the aid of smart devices during examinations. And in social settings, people are often unwilling to have their conversations recorded by their friends wearing smart glasses. To date, the enforcement of these restrictions has been ad hoc, and not surprisingly, easily bypassable.

In this paper, we present a systematic approach for restricted space hosts to remotely analyze and regulate guest device use in the restricted space. Our approach cleanly separates policy from mechanism. Policies on device use are decided by the hosts that control the restricted space. These policies are enforced by a trusted mechanism that executes on the smart guest device. We present an implementation of our mechanism on smart devices equipped with the ARM TrustZone architecture. We also showcase a number of useful policies that show how device use can be regulated in a variety of restricted spaces.

## 1. INTRODUCTION

Over the last few years, we have witnessed an increase in the number and diversity of personal computing devices. These include smart phones, tablets, smart glasses, watches, and a variety of special-purpose embedded devices, such as health monitors and sensors. We can expect these devices to become integral parts of end-users' daily lives, *e.g.*, smart glasses used for prescription vision correction, and watches used as continuous health monitors. As a result of these developments, smart devices will come to be used in a wide variety of settings, such as the home, in social settings, and at school and work. End-users will carry these devices as

they go, and expect them to connect and work with the environments at the places they visit.

As smart devices evolve, their computing power will continue to increase and so will the quality and diversity of peripherals available on them. While this is generally a desirable trend, there are environments in which highly-capable smart devices may be misused to gather and exfiltrate information from the environment, or smuggle unauthorized information into the environment. 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 [44]. In social settings, people may be uncomfortable at the thought of their videos and conversations being recorded by smart devices. In this paper, we introduce 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 corporate 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. And in the restaurant/bar and social settings, enforcement is often informal and left to privacy-conscious patrons or hosts.

Going forward, such ad hoc methods will prove inadequate. First, our increasing reliance on smart devices will make traditional methods difficult to use. For example, it would not be possible to ask an employee or a student with prescription smart glasses to refrain from using the device. In this case, the right solution would be to allow the smart glass to be used as a vision corrector but regulate the use of its camera, microphone or WiFi interfaces. Second, the shrinking form factor and increasing diversity of smart devices will make it hard to identify policy violations. 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 smart watches [46]. And third, at least in the enterprise setting, current trends indicate that employees may in fact be encouraged to use their personal devices in corporate settings. This “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 personal devices comply with the policies of the restricted spaces within which they are used. Such a method would benefit both the *hosts* who own or control the restricted space and the *guests* who use the smart device. Hosts will have greater assurance that smart devices used in their spaces conform to their 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.

In this paper, 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, 3G/4G. In Section 3, we present a number of scenarios where controlling peripherals would be useful.

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.

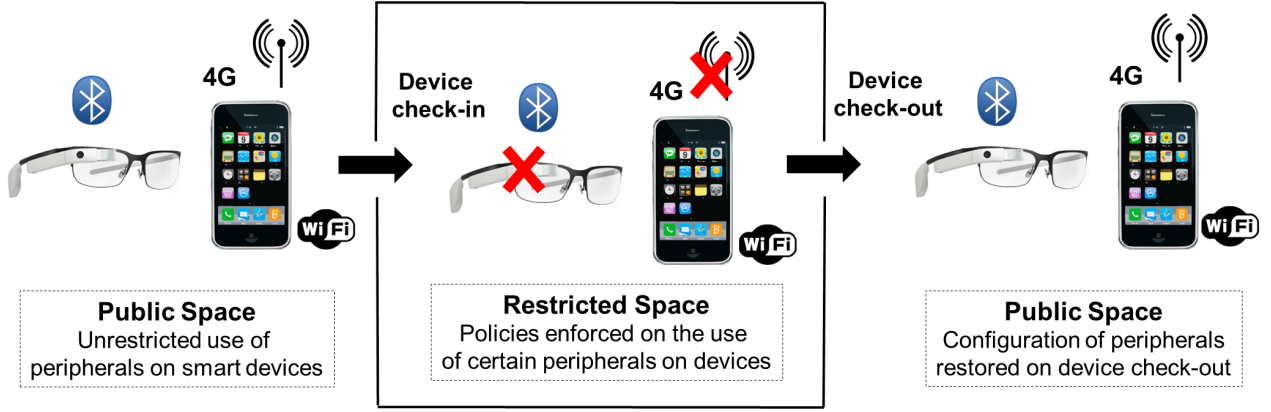
We describe an approach to implement this enforcement mechanism using *remote memory operations* on the guest device. Hosts use these operations to remotely inspect and modify the memory contents of the guest device based on its policies. The host must be able to ensure that these operations are performed 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. Devices that use the ARM TrustZone are now commercially available, with many millions of devices deployed already [10]. 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. It may also be possible to implement our mechanism on virtualized guest devices [6, 8, 23, 25]. We discuss the design of the enforcement mechanism in Section 4.

**Contributions.** To summarize, the main contributions of this paper are the following:

- *Restricted space model.* We introduce the restricted space model for smart personal devices. This model is of independent interest because of the growing number of environments that restrict the use of smart devices (Section 2).
- *Enforcement mechanism.* We present the design of a mechanism for hosts to enforce policies on guest device use. Our mechanism allows hosts to remotely inspect, analyze and control a guest device using a simple interface with two operations: remote memory reads, and remote memory writes. The mechanism also allows guests and hosts to mutually authenticate each other, thereby ensuring that only trustworthy hosts are allowed to make guest modifications (Section 4).
- *Prototype implementation.* We demonstrate a prototype implementation of the mechanism using the ARM TrustZone hardware as a root of trust on guest devices. Our mechanism is minimal, thereby ensuring a small TCB on guest devices (Section 5).
- *Case studies.* We present a number of scenarios of restricted spaces and policies that can be enforced by our mechanism to regulate the use of smart devices (Sections 3 and 5).

## 2. RESTRICTED SPACE USAGE MODEL

We now provide a high-level overview of the restricted space model. Our goal is to motivate the use of some of the



**Figure 1: An overview of our restricted space model.** Guests “check-in” 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 can continue to use its Bluetooth interface to pair with the smart phone, while the phone can use WiFi. When guests leave the restricted space, they “check-out” their devices, which the host then restores to their original configurations.

mechanisms that are described in Section 4, and to describe our threat model.

## 2.1 Overall Workflow

**Check-in.** As depicted in Figure 1, when a guest enters a restricted space in which he wishes to use his devices, he “checks-in” each of his devices during entry. Check-in can happen at a specially-designated kiosk, where the host connects a guest device to its policy server to perform the following tasks:

(1) *Authentication.* The authentication step allows the guest and host to mutually identify each other. We assume that both the guest and the host have cryptographic credentials (*e.g.*, public/private key pairs) that can be 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, for example, as is typically done during SSL/TLS handshakes.

The host’s policies are enforced by a mechanism that executes in the guest. As previously discussed, this mechanism is part of the TCB, and must therefore not be tampered by the guest. We rely on secure hardware on the guest device (the ARM TrustZone) to ensure this property. In particular, we rely on the secure hardware to detect and prevent any attempts to tamper with the TCB on the guest. This can be ensured using a special boot-time protocol implemented atop the secure hardware.

It is important to note that this TCB is *not* the end-user’s usual work environment on the device, *e.g.*, the traditional Android, iOS or Windows environment with apps. Rather, the TCB is an entity that would be created and distributed by

a trusted entity, such as the device vendor, and execute on an isolated environment in the guest device.

(2) *Host analyzes device.* Having attested the TCB, the host leverages it to analyze the device. Our mechanism allows hosts to *remotely fetch memory pages* (and CPU register state) belonging to the guest device. The TCB computes a cryptographic checksum over the memory pages to ensure that they are not tampered with as they are transferred from the guest to the host.

The host uses raw memory pages from the device in two ways. First, it scans memory pages to ensure that the device is free of malicious software, *e.g.*, in the form of malicious apps or kernel modules. Second, it extracts configuration information from the device. This includes the kernel version, the list of peripherals supported by the device, memory addresses of various device drivers for peripherals on the device and the state of these peripherals (*e.g.*, whether a certain peripheral is enabled and its settings). The host can also checkpoint the state of these peripherals so that they can be restored at check-out.

(3) *Host modifies guest’s configuration.* The host modifies the guest device’s configuration to conform to its restricted usage policies. In the example shown in Figure 1, the host’s policy is to disable the camera and the 3G/4G data plan on all devices. The host enforces this policy by creating a sequence of *remote memory writes* that directly modify the memory state of the device. For example, the host could prevent the use of the camera and 3G/4G data plan by unlinking the device drivers corresponding to the camera and the modem. The host uses the configuration information extracted in the previous step to decide the memory addresses that must be modified to unlink these drivers.

We assume that it is the host’s responsibility to ensure that these configuration modifications are not easily bypassable. For example, these modifications may be undone if the user of the guest device can directly modify kernel memory, *e.g.*, by dynamically loading kernel modules or via the `/dev/kmem` interface in the end-user’s work environment. The host must use the inspection phase to identify device configurations that could lead to such attacks, and disallow the use of such devices within the restricted space.

(4) *Host obtains verification token from guest.* After the guest device has been configured, the TCB produces a *verification token* to be transmitted to the host. The verification token is an unforgeable value that encapsulates the set of configuration changes to the device. The TCB computes the token as a checksum over the memory locations that were modified. The token is unforgeable in that only the TCB can re-create its value as long as the device configuration has not been altered, and any malicious attempts to modify the token can be detected by the TCB and the host.

At any point when the device is in the restricted space, the host can request the TCB on the device to send it the verification token. The TCB computes this token afresh, and transmits it to the host,<sup>1</sup> which compares this freshly-computed token with the one obtained during check-in. It can use this comparison to ensure that the guest has not altered the device configurations from the previous step. The verification token is ephemeral, and can be computed afresh by the guest only within an expiration period. In our prototype, the TCB cannot recompute the verification token if the guest device is rebooted, thereby ensuring that end-users cannot undo the host’s configuration changes by simply rebooting the device.

**Check-out.** Once checked-in, the guest device can freely avail of the facilities of the restricted space under the policies of the host. For example, in Figure 1, the smart glass can pair with the smart phone via Bluetooth, while the smart phone can connect to and use the host’s WiFi access point. When the guest exits the restricted space, it must check-out the device, which accomplishes two tasks.

(1) *Host checks guest state.* The host checks guest device’s configuration by requesting the verification token from the device’s TCB to ensure that the configuration has not been altered. If it finds that the verification token does not match the value obtained from the device at check-in, it can detain the device for further inspection, *e.g.*, to determine whether any data from the restricted environment was exfiltrated.

Note that it is not usually possible to differentiate between mismatches that happen because of benign reasons, such as a device reboot, or malicious ones, such as when an end-user intentionally modified the peripheral configurations re-

<sup>1</sup>Of course, this assumes that the host’s policy does not disable all the guest peripherals, and that a communication channel still exists between the host and the guest. If all peripherals are disabled, the host will need physical access to the guest to visually obtain the freshly-computed verification token.

quired by the host. This is because even malicious modifications can be masked by simply rebooting the device, and making the mismatch seem benign. Thus, the host’s policy to deal with mismatches depends upon the sensitivity of the restricted environment. For example, in a federal setting, a detailed forensic examination of the device may be necessary, the possibility of which could deter a malicious guest from bypassing the host’s policy enforcement. As previously discussed, hosts can request the verification token from the device at any time when it is in the restricted space. Hosts can use this feature to frequently check the verification token and narrow down the timeframe of the violation.

(2) *Restoring guest state.* To restore the state of the device, the end-user could simply reboot the device. 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 a clean version of the kernel in persistent storage. Alternatively, the host can restore the state of the guest 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 has the illusion of no network connectivity. However, because we only modify memory and do not actually reset the peripheral, the 3G interface may have accumulated packets, which the app may no longer be able to process when the kernel memory state is restored. Mechanisms such as shadow drivers [58] can enable such “hot swaps” of kernel state, and enable the guest device to continue without rebooting.

## 2.2 Malicious Hosts

Our discussion so far has assumed that hosts are benign. Authentication at check-in ensures that remote access to guest device memory is available only to hosts that the guest trusts. However, it may be possible even for such hosts to misuse the facilities of our mechanism to violate the security and privacy of the guest. For example, a malicious host could use remote memory reads to obtain the guest’s personal data. It could also install a key logger or a backdoor to spy on the guest’s activities. Even if the host is not overtly malicious, it is possible that the configuration changes that it makes will render the guest’s device vulnerable to attacks. This is reminiscent of the 2006 incident when CD DRM software installed by Sony made the systems on which it was installed vulnerable to certain kinds of attacks [36].

It is challenging to protect guest devices from malicious hosts, and *for this paper, we will assume that hosts are benign*. Nevertheless, there are a few defenses that could be used to protect against malicious hosts. To protect personal data, privacy-conscious users could store their data encrypted on disk and in memory, except when it is used. However, this would require apps on the device to be redesigned to be aware of such encrypted data.

To protect against remote writes that maliciously modify the guest device (*e.g.*, by installing a keylogger), the guest could rely on a trusted third party to vet the operations of the host. That is, the guest could forward its memory configuration and the remote operations requested by the host to the trusted third party prior to making any memory modifications. The guest would proceed with the modifications only if the trusted third party certifies that the operations do not install malicious software on the guest device. Alternatively, a simpler solution is to restrict the kinds of remote write operations that a host can initiate. For example, the TCB could enforce that the only remote memory writes available to the host are writing NULL bytes to memory locations that store device driver hooks. This would allow hosts to unlink drivers and disable peripherals, but prevent them from installing malicious software.

### 2.3 Covert Use of Devices

It is certainly possible for a guest to bypass the host’s policies by not declaring a device during check-in. 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 mechanisms developed in this paper. Instead, we assume that traditional methods such as physical security checks are necessary to detect covertly-hidden devices.

### 2.4 Threat Model

Given the discussion in this section, we now summarize our threat model. Each guest device is assumed to be partitioned into two parts: (1) the end-user’s work environment, and (2) the environment that runs the policy enforcement mechanism.

From the perspective of the host, the end-user’s work environment is untrusted and could be malicious. The host must trust the policy enforcement mechanism, but because it executes on the guest device, the host leverages secure hardware, the ARM TrustZone in our case, to bootstrap this trust. Having established trust, the host then leverages the policy enforcement mechanism to securely read and modify the memory state of the work environment. It is the host’s responsibility to inspect the memory state of the work environment to determine whether it is malicious, contains known exploitable vulnerabilities, or allows guests to bypass the configuration changes that the host may make. Once the host determines that the guest’s work environment is acceptable, it can induce changes to the guest’s memory state. Guests are assumed to keep their devices powered on for the duration of their stay in the restricted space, failing which the verification tokens will no longer match. Mismatches may also happen if the guest maliciously modifies the host’s changes.

Note that the work environment may contain zero-day vulnerabilities, such as a newly-discovered buffer overflow in the kernel. The host may not be aware of this vulnerability, but a malicious guest may know of it and have an exploit that leverages it to bypass the host’s policies. Such threats are outside the scope of our work, but the host may protect itself by requiring the guest’s work environment to run a fortified software stack, *e.g.*, as is available on the Samsung Knox platform [10] (the host can check this requirement during the inspection phase). A malicious work environment may also launch a denial-of-service attack, which will prevent the host from communicating with the TCB on the guest device. However, such attacks can readily be detected by the host, which can then prevent the device from entering the restricted space. We therefore exclude such attacks from our threat model.

From the perspective of the guest, the host is assumed to only make benign changes to the guest work environment. However, it may be possible to relax this assumption provided the guest vets all the host’s intended changes with a trusted third party. Similarly, although the host is assumed not to maliciously compromise the guest’s privacy, a guest may choose to encrypt its in-memory and on-disk data to protect it from the host.

## 3. DEVICE ANALYSIS AND CONTROL

The ability to remotely inspect and configure guest devices gives hosts the power to enforce a wide array of policies. In this section, we present a number of applications of such remote device analysis and control.

### 3.1 Kernel Analysis

The host can perform a variety of analyses on the guest device by remotely reading its memory contents.

**Kernel-malware Detection.** The host can analyze memory pages to identify malicious software in the end-user’s work environment. Such *kernel-level rootkits* allow attackers to retain long-term control over infected devices, and are a realistic threat for mobile devices [15, 16]. It is well-accepted by the security community that rootkits cannot reliably be detected by user-space anti-malware tools on the infected device because these tools rely on the services provided by the operating system, which is itself infected. Rootkits can reliably be detected only using external methods, which inspect the infected kernel without relying on its services.

There are a number of ways to detect kernel-level malware by analyzing raw-memory pages. In the past, simple kernel-level rootkits operated by maliciously modifying code pages of the kernel. The host can easily detect such rootkits by checking that all executable code pages of the kernel belong to a whitelist of known-good pages. Modern rootkits work by injecting malicious code into the kernel, and modifying key kernel data structures to force their execution [48]. For example, a rootkit could inject a device driver, and modify

function pointers within the kernel to invoke functions from this driver. More generally, a rootkit can modify an arbitrary kernel data structure to implement its malicious functionality. Such data structure modifications can be used to hide malicious user processes by modifying process lists, disable firewalls by modifying kernel hooks, and weaken cryptographic functions by modifying entropy pools [13].

Prior work [12, 19, 24, 48, 51] has developed a generic approach to detect malicious data modifications by analyzing raw memory pages. The main idea is for the host to externally reconstruct 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 kernel being analyzed, *e.g.*, the sizes, layouts, and fields of every data structure. The host starts the reconstruction process from some well-known entrypoints into the system’s memory, *e.g.*, the addresses of the entities in `System.map`, and recursively traversing kernel data structures. When the traversal process encounters a pointer, it must fetch the data member referenced by the pointer. The host therefore iteratively requests virtual pages from the guest device based upon the needs of this recursive traversal. Having reconstructed 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 pre-determined addresses in the kernel’s code space [48]. Similarly, the host can check that the kernel’s data structures satisfy invariants that must typically hold in an uncompromised kernel [12].

**Reverse-engineering Configurations.** The host can use the memory pages obtained from the guest to identify the version of the kernel, and to determine if security patches have been applied. The host can also use it to check whether the configuration of the kernel disallows certain well-known attack interfaces (*e.g.*, that access to `/dev/kmem` and dynamic module loading are disabled). The recursive traversal method described above can also be used to reconstruct kernel configurations. For example, it can be used to identify addresses at which the functions of a peripheral’s driver are loaded, where they are hooked into the kernel, and the addresses that store memory-mapped peripheral configurations. The host uses this information to create remote write requests that can reconfigure or uninstall peripherals from the device.

**Check Running Processes and Apps.** Once the host has ensured that the kernel is not infected by rootkits, it can use the clean kernel to bootstrap security at the user level. For example, it can require the end-user’s work environment to execute an anti-malware app that relies on the clean kernel to detect sources of infection at user-space, *e.g.*, infected files or apps that reside on the file system. The host can ensure that this anti-malware app is executing on the guest device during check-in by traversing the kernel’s list of running processes. At check-out, it can ensure that the same app

is still executing, *e.g.*, by comparing its process identifier to the value obtained at check-in.

### 3.2 Control over Peripherals

The host can remotely configure peripherals on the guest device by writing to suitable memory locations. We consider various peripherals available on modern devices and describe scenarios where control over them would be useful.

**Camera.** The use of the camera is perhaps the most obvious way for guests to violate privacy. 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. 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 so that the microphone is activated only when a call is placed, and is de-activated otherwise. However, this facility can potentially 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 (*e.g.*, 911 or 112), 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 typically to avoid exfiltrating information outside the enterprise. With our approach, enterprises can allow employees to use the network after inspecting their devices to ensure that they are free from malicious software. 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 [53]) 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 locally on the phone. In such cases, disabling Bluetooth prevents this side channel, but still allows the student to use the glass as a vision-correction device.

## 4. POLICY ENFORCEMENT

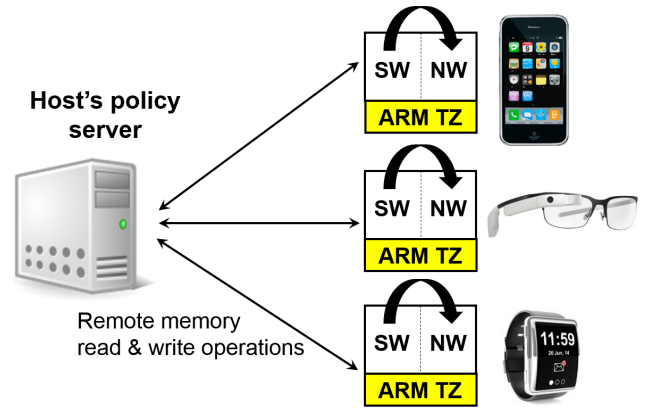
We now present the design of our policy enforcement mechanism. Although the core of our mechanism that relies on remote memory operations is platform agnostic, we describe it in the context of the ARM TrustZone. We leverage the TrustZone’s features to isolate the enforcement mechanism and to bootstrap its security features. We present alternative designs in Section 6.

### 4.1 Background on 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. On TrustZone, the processor can execute 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 can switch 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 performs a context switch to the secure world (via the monitor mode) and freezes the execution of the normal world.

TrustZone can partition 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 run execute arbitrary software, ranging from simple applications to an entire operating system.

A device running ARM TrustZone boots up in the secure world. After the secure world has initialized, it switches to the normal world and boots the operating system there. Most TrustZone-enabled devices are configured to execute a *secure boot* sequence that incorporates cryptographic checks into the secure world boot process [5]. For example, the device vendor could sign the code with its private key, and the



**Figure 2: Overall setup showing communication between the host and guest devices. Guest devices are equipped with ARM TrustZone and execute components of the policy enforcement mechanism (see Figure 3). Abstractly, the goal is to establish a communication channel between the host’s policy server and the enforcement mechanism running in the secure world (SW) of the guest devices. The host leverages the secure world to remotely inspect and modify the normal world’s (NW) memory.**

vendor’s code in the boot ROM would verify 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 TCB. In the rest of this paper, we will assume that our guest devices use the secure boot process.

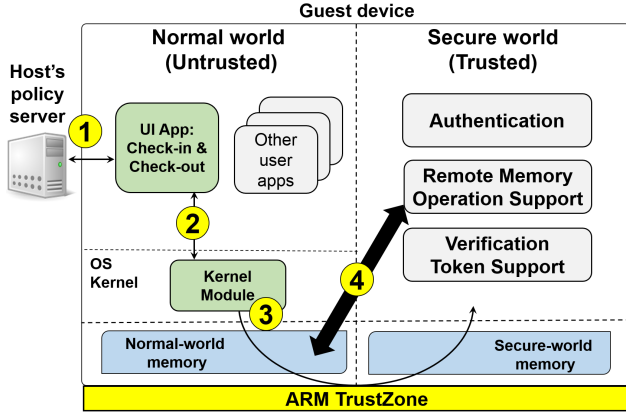
### 4.2 Overall Design

Figure 2 shows the overall setup of our framework. 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. 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 set up such a channel is to configure the secure world to directly communicate with the host. In this case, the secure world would exclusively control a communica-





**Figure 3: Guest device setup showing components of the policy enforcement mechanism.** (1) The host communicates with the UI app on the guest and sends requests to perform remote memory operations. (2) The UI app forwards this request to the supporting kernel module. (3) The kernel module invokes the secure world by performing a world switch. (4) The secure world performs the requested memory operations on the normal world memory on behalf of the host. The components in the normal world, *i.e.*, the UI app and the kernel module are untrusted. The TCB consists of only the components that execute in the secure world.

tions peripheral, say WiFi, and establish a connection with the host without involving the normal world. Thus, the code necessary to support this peripheral must also execute within the secure world and be part of the TCB. With WiFi, for instance, this would mean that several thousand lines from the networking stack would need to execute within the TCB.

In our work, we chose an alternative approach that minimizes the functionality implemented in the TCB. In this approach, the normal world mediates the communication channel between the secure world and the host, and is assigned all the peripherals on the device. Because the normal world is untrusted, the secure world and the host include cryptographic checksums with each message transmitted on the channel, and sanity-check the messages before processing them. The secure world itself executes a TCB that performs just three key operations, and the bare-minimum code needed to support these operations:

- (1) Mutual authentication (Section 4.3),
- (2) Support for remote memory operations (Section 4.4),
- (3) Support for verification tokens (Section 4.5).

Guest devices are therefore set up as shown in Figure 3. Within the normal world, the end-user interacts with the host as well as with the secure world via a user-level app (called the UI app). This app serves as the end-user’s interface to the host, allowing him to perform operations such as check-in and check-out of the device. The app interacts with the components in the secure world via a kernel module. The

Let host’s public/private keypair be  $\text{PubH}, \text{PrvH}$ .  
 Let guest’s public/private keypair be  $\text{PubG}, \text{PrvG}$ .

1. **Guest**  $\rightarrow$  **Host**:  $\text{PubG}, \text{Cert}(\text{PubG})$
2. **Host**  $\rightarrow$  **Guest**:  $\text{PubH}, \text{Cert}(\text{PubH})$
3. Guest and host verify  $\text{Cert}(\text{PubH})$  and  $\text{Cert}(\text{PubG})$
4. **Host**  $\rightarrow$  **Guest**:  $M, \text{Enc}_{\text{PrvH}}(M)$  (*i.e.*, host signs  $M$ ), where  $M$  is  $\text{Enc}_{\text{PubG}}(k_s, \text{timestamp})$
5. Guest verifies host’s digital signature, decrypts  $M$  to obtain  $k_s$ , and checks timestamp

**Figure 4: Mutual authentication and establishment of session key  $k_s$ .**

host sends requests to perform remote memory operations on the guest device to the app, which forwards this 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.

We do not place any restrictions on how the host communicates with the guest. Thus, the host’s policy server could be hosted on the cloud and communicate with the guest device over WiFi or 3G. Alternatively, the host could install physical scanners at a kiosk or on the entryway to the restricted space. The guest devices would use Bluetooth, NFC, or their USB interface to pair with the scanner and use it to communicate with the host’s policy server.

One of the key features of our approach is that the core mechanisms that run on the TCB in the guest device are *platform-agnostic* and *policy-agnostic*. Rather, they present a narrow read/write interface that hosts can suitably use to enforce policy. All the complex tasks of device analysis and deciding policy are shifted to the host. Thus, in principle, we can use the same TCB mechanisms on guest devices that run Android, iOS and Windows. Hosts would have separate modules to analyze and control each kind of guest platform.

### 4.3 Authentication

Before accepting any remote memory requests, the guest and the host mutually authenticate each other. 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  $\text{PrvG}$  in its secure world, thereby protecting it from the untrusted normal world.

Authentication proceeds in a manner similar to TLS handshakes (Figure 4). 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 an secure channel. Again, this resembles session key establishment in TLS. The major difference is that  $k_s$  is only used to protect the integrity of messages transmitted between the guest and the host and not its confidentiality. The key  $k_s$  is stored in secure world memory, and is therefore invisible to the normal world. If the guest is rebooted,  $k_s$  is erased from memory.



## 4.4 Remote Memory Operations

**Remote Reads.** The host inspects and modifies the guest device’s configuration via remote memory operations. 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 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 over the body of the request using the key  $k_s$  negotiated during the authentication phase.

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 is iterative, with the host requesting more pages from the guest based upon the results of the analysis.

Note that in our case, 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 check the integrity of messages communicated over this channel, thereby preventing any attempts by the normal world kernel to maliciously modify these messages. The normal world may drop messages and cause a denial-of-service attack; however, such attacks are outside our threat model (see Section 2.4). The host can therefore reliably obtain the memory pages of the normal world to enable the kinds of analyses described in Section 3.1. Although the communication between the host and the secure world is integrity-protected using HMACs, the communication is not confidential, and is therefore not encrypted. Thus, a malicious normal world kernel can potentially snoop on the requests from the host to fetch pages and attempt to remove the source of infection to avoid being detected. However, this would also 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 write requests. Each write request is 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 request to the secure world, which verifies the integrity of the message using its 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 values match, then the secure world replaces their values with  $val_i$ .

Note that 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 error code denoting a failure.

A remote write request can fail if the value stored at the virtual address  $vaddr_i$  does not match the value  $old-val_i$ . This problem arises in our design 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 either have updated the value at the virtual address  $vaddr_i$ .

If a remote write operation fails, the host repeats the operation until it succeeds. That is, it refetches pages from the guest, analyzes them, and creates a fresh write request. In theory, it is possible that the host’s write requests will fail *ad infinitum*. However, for the setting that we consider, write operation failures 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 write request can also fail 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. Therefore, we do not observe failures due to a failure to resolve  $vaddr_i$ s.

It is possible to completely avoid such problems by design if we complete both the read and write operations in 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 already discussed in Section 4.2, we decided against this design because it has the unfortunate consequence of bloating the size of the TCB. Thus, we make the practical design tradeoff of minimizing

the functionality of the TCB while allowing the rare remote write failure to happen.

#### 4.5 Verification Tokens

The host receives a verification token from the secure world upon successful completion of a remote write operation. A verification token  $\text{VTok}[r]$  is the value:

$$\text{VTok}[r] = r || \text{MemState} || \text{HMAC}_{k_s}[r || \text{MemState}]$$

where  $\text{MemState}$  is  $\langle \text{vaddr}_1, \text{val}_1 \rangle || \dots || \langle \text{vaddr}_n, \text{val}_n \rangle$ , the set of  $\text{vaddr}_i$  modified by the remote write, and the new values  $\text{val}_i$  at these locations. The token  $\text{VTok}[r]$  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 configuration changes made by the remote write, either maliciously or by turning off the guest device. To do so, the host obtains a verification token  $\text{VTok}[r_{\text{checkin}}]$  upon completion of checkin, and stores this token for validation. During checkout, the host requests a validation token  $\text{VTok}[r_{\text{checkout}}]$  from the guest over the same virtual memory addresses. The secure world accesses each of these memory addresses and computes the verification token with  $r_{\text{checkout}}$  as the nonce. The host can compare the verification tokens  $\text{VTok}[r_{\text{checkin}}]$  and  $\text{VTok}[r_{\text{checkout}}]$  to determine whether there were any changes to the values stored at these memory addresses.

Assuming a good source of randomness, the nonces  $r_{\text{checkin}}$  and  $r_{\text{checkout}}$  ensure the freshness of the tokens  $\text{VTok}[r_{\text{checkin}}]$  and  $\text{VTok}[r_{\text{checkout}}]$ . The use of the  $k_s$  to compute the HMAC in the verification token ensures that the token is only valid for the duration of the session, *i.e.*, until check-out 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. This ensures that any attempts, either malicious or benign, to undo the configuration changes induced by the remote writes at check-in will be detected by the host.

### 5. EVALUATION

We implemented our policy enforcement mechanism atop a Freescale i.MX53 Quick Start Board as our guest device. This TrustZone-enabled board 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.

Name of Component	LOC (C code)
<b>Secure World</b>	
(1) Memory manager	1,381
(2) Authentication	1,285
(3) Reads/writes/tokens	305
(4) PolarSSL files	
• SHA1+HMAC	861
• X509	877
• RSA	2,307
<b>Normal World</b>	
(1) Kernel module	93
(2) UI app	72

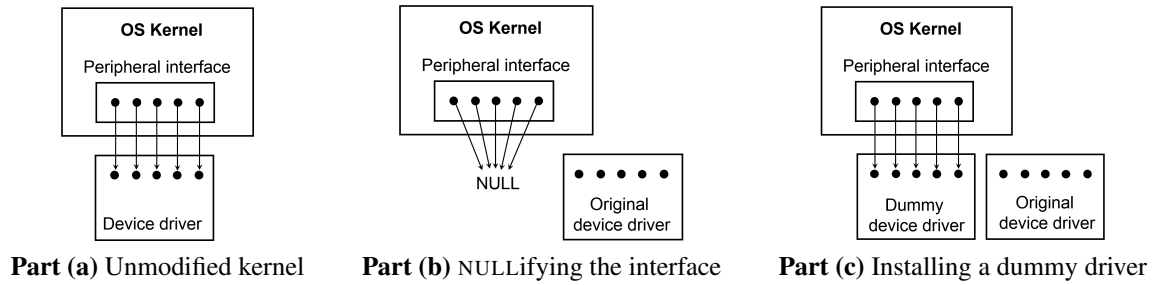
**Table 1: Sizes of components executing on the guest.**

We built a bare-bones runtime environment for the secure world, just enough to support the components shown in Figure 3. 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 PolarSSL (v1.3.9) [3]. Excluding the cryptography library, our TCB in the secure world consists of about 3,000 lines of C code, including about 250 lines of inline assembly. Table 1 shows the sizes of various components. We used PolarSSL’s implementation of SHA1 and HMACs, RSA and X509 certificates. As shown in Table 1, the files implementing these components alone comprise only about 4,000 lines of code. In addition to these TCB components, we built the kernel module and the UI app (written as a simple, native daemon) for the normal world, together comprising 165 lines of code.

We also implemented a host policy server that authenticates guest devices, and performs remote memory read and write operations. We conducted experiments to showcase the utility of remote reads and writes to enforce the host’s policies on the guest. In our prototype, 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, a survey by Petroni and Hicks [48] shows that over 95% of all rootkits that modify kernel data operate in similar ways.

We were able to detect this rootkit on the host by remotely fetching and analyzing memory pages from the guest. We requested the guest to send all the pages containing the `init`, `text` and `data` sections of kernel memory, and analyzed these pages. 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



**Figure 5: Uninstalling a peripheral by writing to kernel memory.** Part (a) shows how peripheral device drivers are linked to the kernel. Part (b) shows how to uninstall the peripheral by making the kernel’s device interface point to NULL bytes. Part (c) shows how to uninstall the peripheral by unlinking the original driver and instead linking a dummy driver.

address is not included, *e.g.*, as happens if the entry for the `close` system call is modified, it raises an error. In prior work [12], we have built a tool to recursively traverse kernel data structures from memory dumps. This tool can also be used with the memory pages obtained by the host to detect more sophisticated rootkits.

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.

**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) operating system. Thus, while remote writes to this device do not enjoy the security properties described in Section 4, they allow us to evaluate the ability to uninstall a variety of peripherals from a running guest device.

We adopted two broad strategies to uninstall peripherals. As shown in Figure 5(a), 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 in turn communicates with the peripheral.

Peripheral	Method Used	Size (bytes)	Device
USB (webcam)	NULLify interface	104	i.MX53
USB (webcam)	Dummy driver	302	i.MX53
Camera	NULLify interface	140	Nexus
Camera	Dummy driver	212	Nexus
WiFi	Dummy driver	338	Nexus
3G (Data)	Dummy driver	252	Nexus
3G (Voice)	Dummy driver	224	Nexus
Microphone	Dummy driver	184	Nexus
Bluetooth	Dummy driver	132	Nexus

**Table 2: Peripherals uninstalled using remote write operations to a guest device.** Also shown are the method used to uninstall the peripheral (see Figure 5), the number of bytes modified or added by the remote write, and the guest device on which we performed the operation.

Our first strategy to uninstall the peripheral was to simply set the function pointers in the peripheral’s interface to NULL (Figure 5(b)). 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 that this approach generally makes the guest unstable. For example, except in two cases, NULLifying the peripheral interface resulted in a system crash (Table 2).

We therefore adopted the approach shown in Figure 5(c), which should work with any peripheral. In this approach, we write a dummy driver for the peripheral, *e.g.*, one that simply returns a suitable error code rather than communicating with the peripheral, and link it with the kernel in place of the original driver. Note that the host must introduce new code (*i.e.*, the dummy driver) into the guest, so this approach may be unpalatable to some guests, and may even make the guest more vulnerable if the dummy driver contains bugs (*e.g.*, [36]). Nevertheless, with this approach we were successfully able to uninstall all the peripherals that we tried. As discussed in Section 2.2, guests can employ a trusted third party to ensure the trustworthiness of the dummy driver.

<b>USB</b>	<i>MobileWebCam</i>	<i>Camera ZOOM FX</i>	<i>Retrica</i>	<i>Candy Camera</i>	<i>HD Camera Ultra</i>
<i>Passive</i>	APPERRMSG	APPERRMSG	ANDROIDERRMSG	APPERRMSG	ANDROIDERRMSG
<i>Active</i>	APPERRMSG	APPERRMSG	APPERRMSG	APPERRMSG	APPERRMSG
<b>Camera</b>	<i>Camera for Android</i>	<i>Camera MX</i>	<i>Camera ZOOM FX</i>	<i>HD Camera for Android</i>	<i>HD Camera Ultra</i>
<i>Passive</i>	ANDROIDERRMSG	APPERRMSG	APPERRMSG	ANDROIDERRMSG	ANDROIDERRMSG
<i>Active</i>	BLANKSCREEN	APPERRMSG	ANDROIDERRMSG	BLANKSCREEN	BLANKSCREEN
<b>WiFi</b>	<i>Spotify</i>	<i>Play Store</i>	<i>YouTube</i>	<i>Chrome Browser</i>	<i>Facebook</i>
<i>Passive</i>	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN
<i>Active</i>	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN
<b>3G (Data)</b>	<i>Spotify</i>	<i>Play Store</i>	<i>YouTube</i>	<i>Chrome Browser</i>	<i>Facebook</i>
<i>Passive</i>	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN
<i>Active</i>	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN	LOSTCONN
<b>3G (Voice)</b>	<i>Default call application</i>				
<i>Passive</i>	APPERRMSG: Unable to place a call				
<i>Active</i>	APPERRMSG: Unable to place a call				
<b>Microphone</b>	<i>Audio Recorder</i>	<i>Easy Voice Recorder</i>	<i>Smart Voice Recorder</i>	<i>Sound and Voice Recorder</i>	<i>Voice Recorder</i>
<i>Passive</i>	APPERRMSG	APPERRMSG	APPERRMSG	APPERRMSG	APPERRMSG
<i>Active</i>	EMPTYFILE	EMPTYFILE	EMPTYFILE	EMPTYFILE	EMPTYFILE

**Note:** 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 3: Results of robustness experiments for both the passive and active settings.**

Table 2 shows the set of peripherals that we uninstalled, the method that we used to uninstall the peripheral, the size of the write operation reported as the number of bytes that we had to modify or introduce into the kernel, and whether the operation was performed on the i.MX53 or the smart phone. We were able to uninstall the USB on the i.MX53 and the camera on the phone by NULLifying the peripheral interface. For the other peripherals, we introduced dummy drivers. We also used the dummy driver approach 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 data transmission and that of only disabling calls. As this table shows, it is possible to uninstall peripherals without crashing the operating system or user apps by just modifying a few hundred bytes of memory on the running device.

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. 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 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.

## 6. DISCUSSION

The design of our enforcement mechanism relies on the ARM TrustZone to isolate the TCB and to provide a root of trust on the guest device. While devices featuring the ARM TrustZone hardware are becoming increasingly available (millions of deployed devices, according to the Samsung Knox team [10]), we consider alternatives that can be used by hosts to regulate guest devices.

**Virtualization.** Historically, on x86 server-class machines, virtualization is by far the most popular technique to perform introspection [20, 33]. There have been impressive recent efforts to build virtualization infrastructure for smart phones [6, 8, 23] and other devices based on the ARM architecture [25].

It is possible to regulate the use of peripherals on virtualized smart devices using virtual machines (VMs) to implement various persona. For example, for an employee to use his personal device at work, he may be required to download a work VM from enterprise. The virtual peripherals in

this work VM can be pre-configured to reflect the enterprise policy. Thus, if the enterprise disallows the use of the camera, the work VM is simply configured to not export a virtual camera peripheral to apps executing within the VM. Virtualized platforms also support remote memory operations—the hypervisor simply reads/modifies the virtual memory pages belonging to a VM to perform the operations. Thus, the enterprise can even dynamically reconfigure its work VM via such operations.

However, in virtualized solutions, the TCB typically includes the hypervisor. Although prior work has developed small hypervisors containing only a few thousand lines of code (*e.g.*, Nova [55]), modern production hypervisors are must support different virtualization modes, guest quirks, and hardware features. This makes them large and complex, and has the unfortunate consequence of bloating the TCB. Moreover, without a hardware root of trust, it is hard to establish whether the guest has maliciously bypassed the host’s enforcement. For example, the employee in the above example could suspend the work VM, and switch to a different VM that allows the use of the camera. The hypervisor must include additional functionality to prevent the employee from suspending the work VM, which further increases the size of the TCB.

TCB size-concerns aside, we also feel that virtualization may not be applicable to large majority of personal computing devices. Many classes of smart devices such as smart glasses and watches are personal, single-user devices, and specialized in their function. Judging by their typical use-cases, virtualization may be an overkill for most of these devices except those at the highest end of the form factor (*e.g.*, enterprise-class smart phones and tablets). This reduces the motivation to invest the manpower needed to virtualize them.

**Alternative Network Interfaces.** As an alternative to virtualization, it may be possible to use hardware interfaces to perform remote memory operations on a guest system. Such interfaces were investigated for the server world by the high-performance computing community to perform remote DMA as a means to bypass the performance overheads of the TCP/IP stack [1, 2]. This work has since been repurposed to perform kernel malware detection [49] and remote repair [17]. These systems use a PCI-based co-processor on the guest system 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, popularly called the Firewire. The host can leverage the Firewire interface on a guest device to directly read and modify its memory pages. However, a large majority of small form-factor mobile devices available today are not equipped with the Firewire interface. For instance, to our knowledge, except for some laptop computers, most other mobile devices do not have a Firewire port. Another possibility is to use the JTAG interface, defined in

the IEEE 1149.1 standard by the Joint Test Action Group. Via a small number of dedicated pins on the chip, this interface allows testing functions such as reading and writing to the memory and registers of the chip. However, the JTAG is primarily used as a debugging aid.

The main drawback of these alternative network interfaces typically cannot authenticate the credentials of the host that initiates the memory operation. This shortcoming can be exploited by a malicious host to compromise the security and privacy of the guest. Moreover, to configure guest devices via these network interfaces, the host must physically plug into the network interfaces of these devices. Thus, these interfaces are best used when the guest can physically authenticate the host and trust it to be benign.

**OS-based Mechanisms.** Operating systems can be modified to dump the contents of their memory. For example, on Android, the LiME framework [37] and similar tools [35, 56, 59] can be used to acquire memory from a running system. However, without trusted hardware, the host has no way to check the integrity of these memory dumps.

Finally, it is possible to regulate the use of guest devices by requiring the user’s work environment to run a security-enhanced operating system. The Android Security Modules (ASM) framework [38], for instance, introduces a set of security hooks in Android. These hooks consult a security policy, installed as a user-space app, to enforce fine-grained control over the device. As an example, policies in the the ASM system can be used to create persona that can determine the set of apps, files, and peripherals that are visible to the end-user. This approach works at a higher level of abstraction than remote memory operations, and therefore has the benefit of offering greater visibility into application-level context and finer-grained control over apps.

However, as with virtualization, the approach presents a large TCB because the security-enhanced operating system must be trusted. Moreover, the security mechanism itself is closely integrated the operating system. In contrast, in approaches that use remote memory operations, the mechanism is agnostic to the operating system executing on the guest device. The burden of analyzing memory and determining which locations to modify is left to the host, thereby allowing a minimal, platform-agnostic TCB to execute on the guest.

## 7. RELATED WORK

**TrustZone-based Frameworks.** Since the introduction of the TrustZone technology, a number of projects have used it to build novel security applications. Closest to our work is TrustDump [57], which proposes a TrustZone-based mechanism to reliably acquire memory pages from the normal world of a device. While similar in spirit to our remote reads mechanism, the focus of the TrustDump paper is to seek an alternative to virtualized solutions for memory in-

tropection with the primary use-case being malware detection. Unlike our work, TrustDump is not concerned with restricted spaces, authenticating the host, or remotely configuring guest devices.

Samsung Knox [10] and the SPROBES project [34] leverage TrustZone to protect the normal world operating system in real-time from kernel-level rootkits. These projects harden the normal world operating system 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 rootkit infections. In our work, remote reads allow the host to detect infected devices, but we do not attempt to provide real-time protection from malware. As discussed in our threat model (Section 2.4), our work can be used in conjunction with Knox to improve the security of the normal world, and remove certain assumptions from our threat model.

TrustZone has also been used to improve the security of user applications. Microsoft’s TLR project [50] and Nokia’s ObC project [41] leverage TrustZone to provide a secure execution environment for user applications, even in the presence of a compromised kernel. Other applications of the TrustZone include ensuring trustworthy sensor readings from peripherals [43] and securing mobile payments [4].

**Enterprise Security.** With the growing emphasis on the BYOD trend, a number of projects have developed enterprise security solutions for smart devices. Many of these projects have focused on enabling multiple persona (*e.g.*, [8, 18, 38]) or enforcing fine-grained mandatory access control policies on smart devices (*e.g.*, [18, 38, 54, 61]). There has also been significant work in enforcing fine-grained access control policies based on the context in which the device is being used (*e.g.*, [11, 21, 22, 26, 45, 47]).

There are three points of difference between these projects and ours. First, they typically exercise their policies at the level of user apps and not peripherals. We view our work as being synergistic with these efforts, complementing the app-level control that they provide. Second, most of these techniques are typically implemented by enhancing the runtime platform on the guest device (typically Android). In contrast, in our work the policy enforcement mechanism that executes on the guest is platform-agnostic because it works at the level of memory reads and writes. The platform-specific portion (*i.e.*, analyzing memory images and designing the write operations) are performed at the host. This approach keeps the TCB that executes on the guest minimal in functionality. Third, a minor point of difference is that these prior projects are not integrated with secure hardware (although they potentially can be extended to leverage the security features of the ARM TrustZone).

**App Security.** Starting with the TaintDroid project [29], a number of researchers have demonstrated that many popular apps exfiltrate sensitive user data from smart devices. When

combined with the fact that a significant fraction of apps (at least on Android) are over-privileged [9, 30], and that end-users are poor at understanding the meaning of app permissions [32, 42], smart devices can turn into spying devices even without the malicious intent of end-users. Such apps can leverage the increasing array of sensors on modern smart devices in novel and dangerous ways [52, 60]. These threats will amplify in the future as we see an increasing number of augmented reality apps (*e.g.*, on wearable devices such as smart glasses) 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 [31] 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 [39, 40]. Our work can complement these efforts by giving hosts the ability to directly 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 [10], as previously discussed, provides the ability to detect certain classes of kernel-level rootkits in real time. MOCFI [27] enhances the mobile operating system by enforcing control-flow integrity properties, thereby mitigating the effect of attacks such as buffer overflow-based exploits. Airbag [62] 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 [28] 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.

## 8. CONCLUSIONS AND FUTURE WORK

We see this paper as contributing two main conceptual advances. The first is the notion of restricted spaces. As argued in the paper, the increasing ubiquity and capability of smart devices has 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.

The second conceptual contribution of this paper is the design of a mechanism for hosts to remotely inspect and control guest devices. We showed that this can 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.

While we believe that we have demonstrated the technical feasibility of our design via a prototype implementation and case studies, a number of issues must be addressed to make it a practical solution. The foremost such concern is end-user willingness to subject their devices to regulation in restricted spaces. It is possible that many users would simply choose not to use their devices within the restricted space than give the host control over their devices. 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.

Even if end-users were willing to subject their devices to regulation, it is unclear to what extent they will trust the host’s control over their device. Again, we feel that this is largely a matter of user-perception. Users already place implicit trust in app writers and market vendors when they download apps on their smart devices. Yet, it is now well-known that apps routinely violate this implicit trust and compromise user privacy by sending sensitive user data to unauthorized third parties.

Thus, a practical deployment of our solution would also likely require supporting trust infrastructure, *e.g.*, analysis tools that can vet the host’s operations or present them to the end-user in a human-readable format for approval. We plan to explore such user-interaction and user-perception issues in future work.

## REFERENCES

- [1] The Infiniband trade association. <http://www.infinibandta.org>.
- [2] Mellanox technologies Inc. <http://www.mellanox.com>.
- [3] PolarSSL: Straightforward, secure communication. <https://polarssl.org>.
- [4] Proxama. <http://www.proxama.com/products-and-services/trustzone>.
- [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](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://goo.gl/D3uJJr>.
- [7] N. Anderson and V. Strauss. Cheating concerns force delay in SAT scores for South Koreans and Chinese. In *Washington Post*, October 2014. <http://goo.gl/6n45Gx>.
- [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 (SOSP)*, 2011.
- [9] K. Au, B. Zhou, J. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
- [10] 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 (CCS)*, 2014.
- [11] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen. Context-aware usage control for Android. In *International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.
- [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 and Privacy (Oakland)*, 2007.
- [14] A. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading privacy for application functionality on smartphones. In *HotMobile*, 2010.
- [15] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus energy tradeoffs in host-based detection of mobile malware. In *International Conference on Mobile Systems, Applications, and Services ( MobiSys)*, 2011.
- [16] J. Bickford, R. O’Hare, A. Baliga, V. Ganapathy, and L. Iftode. Rootkits on smart phones: Attacks, implications and opportunities. In *HotMobile*, 2010.
- [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 (ICAC)*, 2004.
- [18] 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.
- [19] 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 (CCS)*, 2009.
- [20] P. M. Chen and B. Noble. When virtual is better than real. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.
- [21] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *International Information Security Conference (ISC)*, 2010.
- [22] M.J. Covington, P. Fogla, Zhiyuan Zhan, and M. Ahamad. A context-aware security architecture for emerging applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2002.
- [23] L. P. Cox and P. M. Chen. Pocket hypervisors: Opportunities and challenges. In *HotMobile*, 2007.
- [24] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium*, 2012.
- [25] C. Dall and J. Nieh. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
- [26] M. Damiani, E. Bertino, B. Catania, and P. Perlasca. GEO-RBAC: A spatially aware RBAC. *ACM Transactions on Information and Systems Security*, 10(1), 2007.
- [27] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

- [28] B. Davis and H. Chen. Retroskeleton: Retrofitting Android apps. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.
- [29] 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 (OSDI)*, 2010.
- [30] A. Felt, E. Chin, K. Greenwood, and D. Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.
- [31] A. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Workshop on Hot Topics in Security (HotSec)*, 2012.
- [32] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Symposium on Usable Privacy and Security (SOUPS)*, 2012.
- [33] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, 2003.
- [34] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the TrustZone architecture. In *IEEE Mobile Security Technologies Workshop (MoST)*, 2014.
- [35] Google. Using DDMS for debugging. <http://developer.android.com/tools/debugging/ddms.html>.
- [36] J. A. Halderman and E. W. Felten. Lessons from the Sony CD DRM episode. In *USENIX Security Symposium*, 2006.
- [37] A. P. Heriyanto. Procedures and tools for acquisition and analysis of volatile memory on Android smartphones. In *11th Australian Digital Forensics Conference*, 2013.
- [38] S. Heuser, A. Nadkarni, W. Enck, and A. R. Sadeghi. ASM: A programmable interface for extending Android security. In *USENIX Security Symposium*, 2014.
- [39] 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.
- [40] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *IEEE Symposium on Security and Privacy (Oakland)*, 2013.
- [41] K. Kostianen, J. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2009.
- [42] 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 (UbiComp)*, 2012.
- [43] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.
- [44] M. McGee. Glass almanac – New website tracks “Glasshole-Free Zones” – businesses that have banned Google Glass, March 2014. <http://goo.gl/h9uew4>.
- [45] 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 (ASIACCS)*, 2014.
- [46] 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 (FC)*, 2014.
- [47] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically-rich application-centric security in Android. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.
- [48] N. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.
- [49] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: A coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.
- [50] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
- [51] K. Saur, M. Hicks, and J. S. Foster. C-Strider: Type-aware heap traversal for C, May 2014. University of Maryland Technical Report.
- [52] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2011.
- [53] 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.
- [54] S. Smalley and R. Craig. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS)*, 2013.
- [55] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *European Conference on Computer Systems (EuroSys)*, 2010.
- [56] A. Stevenson. Boot into recovery mode for rooted and un-rooted Android devices. <http://androidflagship.com/605-enter-recovery-mode-rooted-un-rooted-android>.
- [57] 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 (ESORICS)*, 2014.
- [58] M. M. Swift, M. Annamalai, B. N. Bershad, and H. N. Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4), 2006.
- [59] 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.
- [60] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2013.
- [61] X. Wang, K. Sun, Y. Wang, and J. Jing. DeepDroid: Dynamically enforcing enterprise policy on Android device. In *Network and Distributed System Security Symposium (NDSS)*, 2015.
- [62] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. AirBag: Boosting smartphone resistance to malware infection. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

All URLs in references were last verified on January 28, 2015.
--