Regulating ARM TrustZone Devices in Restricted Spaces

Abstract

Smart personal devices equipped with a wide range of sensors and peripherals can potentially be misused in various environments. They can be used to exfiltrate sensitive information from enterprises and federal offices or be used to smuggle unauthorized information into classrooms and examination halls. One way to prevent these situations is to regulate how smart devices are used in such ***restricted spaces****. In this paper, we present an approach that robustly achieves this goal for ARM TrustZone-based personal devices. In our approach, restricted space hosts use remote memory operations to analyze and regulate guest devices within the restricted space. We show that the ARM TrustZone allows our approach to obtain strong security guarantees while only requiring a small trusted computing base to execute on guest devices.*

1. Introduction

Smart personal computing devices, such as phones, tablets, glasses, watches, 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 [5] or surreptiously collude with peers [33] 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 [31], 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 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 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]

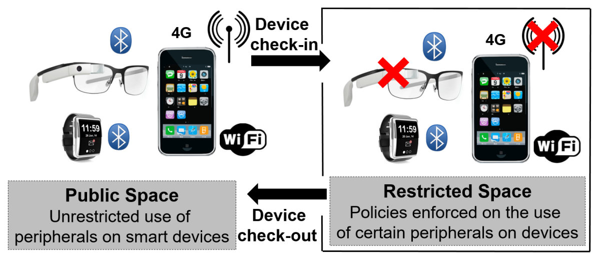
Prior work on this problem (*e.g.,* [13, 24, 32, 35, 40, 41]) has typically assumed that guest devices are benign. The guest device is outfitted with a security-enhanced software stack that is designed to accept and enforce policies supplied by restricted space hosts. A host must trust the software running 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, a malicious guest device with a suitably-modified software stack can easily bypass policy enforcement.

In this paper, we make the following advances:

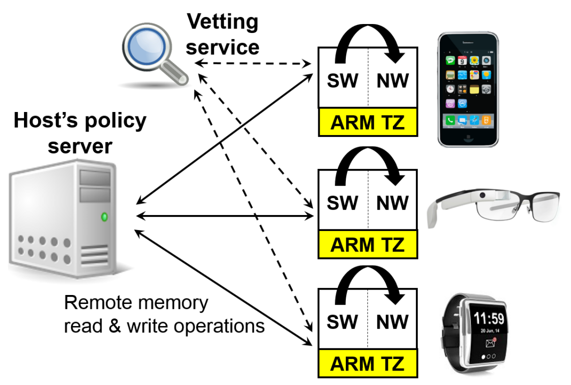
① *Use of trusted hardware.* We leverage the ARM TrustZone [3] on guest devices to offer provable security guarantees. In particular, a guest device can use the ARM TrustZone to produce *verification tokens*, which are unforgeable cryptographic entities that establish to a host that the guest is policy-compliant. Malicious guest devices, which may have violated the host's 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 [8], 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. We show that the use of remote memory operations 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. Com-

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


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

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 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.



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 Section 4. At check-in, the host's policy server leverages the secure world to remotely inspect and modify normal world (NW) memory.

Figure 1: An overview of the entities of our restricted space model and the setup of guest devices.

bined with the 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 *Vetting service.* The two advances above benefit hosts, but guests may be uncomfortable with the possibility of hosts accessing and modifying raw memory on their devices. We therefore introduce a vetting service, trusted and configurable by guests, that allows them to check the safety of the host's memory operations before performing them on the devices. The vetting service ameliorates guests' privacy 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 with a few simple sanity checks, the vetting service allows guests to ensure the safety of the host's remote memory operations.

2. Restricted Spaces

We now 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 key features.

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 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 context switches to the secure world (via monitor mode) and freezes execution of the normal world.

The ARM 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 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 [3, §5.2.2]. For example, the device vendor could sign the code with its private key, and the 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 trusted computing base (TCB). In this paper, we assume that guest devices use secure boot.

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 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 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, 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.

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 Section 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 can directly modify 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 could 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 can 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.

At any point when the device is in the restricted space, the host can request the device to send it the verification token. 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 can use this comparison to ensure that the guest has not altered the normal world memory updates from the previous step. 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. However, in Section 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 can freely avail of the facilities of the restricted space under the policies of the host. For example, in Figure 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:

[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.

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 the device at check-in if the host's memory modifications have been maliciously altered or if the device was rebooted without using REM-suspend. 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 can 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 [48] can possibly enable such "hot swaps" of kernel state and avoid a device reboot.

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 [8] or MOCFI [20]) that implements defenses for common classes of attacks. The host can 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 Section 7). 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 [8] 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.

3. Applications of Remote Memory Operations

We now discuss how hosts can use remote memory read and write operations to analyze and control guest devices.

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

① *Retrieving configuration information.* The host can determine the kernel version by inspecting code pages, thereby also allowing it to check if the guest has applied recommended security patches. The host can compare a hash of each kernel code page against a whitelist, *e.g.,* of code pages in approved Android distributions, to ensure that the normal world is free of malicious kernel code [29, 43]. Additionally, the host can 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 can identify 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 can use the recursive memory snapshot traversal technique described below. The host can use this information to design the set of memory updates that reconfigure the device to make it policy-compliant.

② *Detecting malicious data modifications.* Rootkits can achieve malicious goals by modifying key kernel data structures [10, 36, 37]. 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 [10, 36].

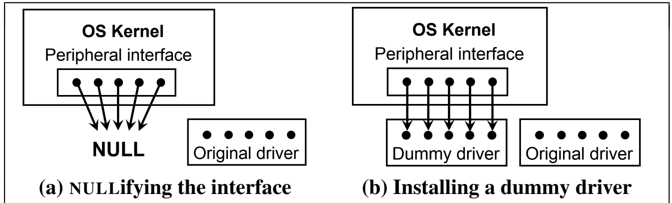
We now describe a generic approach, developed in prior work [9, 14, 18, 37], that hosts can use to detect such malicious data modifications by analyzing the normal world's memory snapshot. The main idea is to recursively traverse the memory snapshot and reconstruct a view of the kernel's data structures, 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. 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 [37]. Similarly, the host can check that the kernel's data structures satisfy invariants that typically hold in an uncompromised kernel [9]. We do not further elaborate on specific rootkit detection policies because they are orthogonal to our focus.

A malicious or 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 are based on policy-enforcing normal world kernels (*e.g.,* [13, 24, 32, 35, 40, 41]) can also benefit from our approach to establish normal world kernel integrity to hosts.

We have restricted our discussion to an analysis of 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 (see Section 5).

To ensure user-space security, hosts can leverage the normal world kernel after establishing that it is benign. 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



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

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 can 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 can be 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.

We adopted two broad strategies to update device drivers:

① *Nullifying interfaces (Figure 2(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 (Section 6) that this approach can crash the device if the kernel expects non-NULL pointers.

② *Dummy drivers (Figure 2(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 [11].

[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.

Dummy drivers can 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.

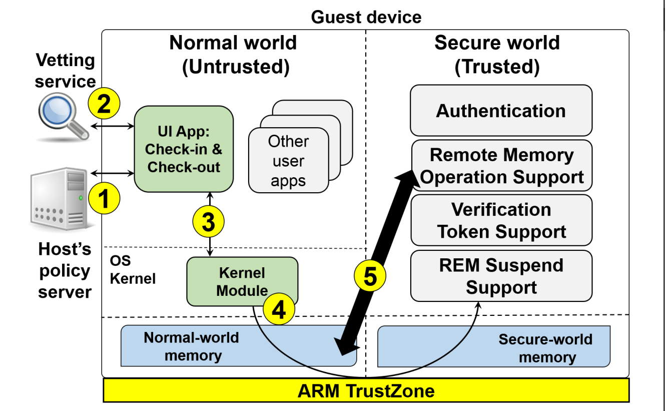
4. Design of the Policy Enforcement Mechanism

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 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 (Section 4.1), remote memory operations (Section 4.2), verification tokens (Section 4.3), and REM-suspend (Section 4.4).

Guest devices are therefore set up as shown in Figure 3. 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 (Section 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 with the scanner and use it to communicate with the host.



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

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 4: Mutual authentication and establishment of k_s.

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.

4.1. Authentication

The host and guest device begin by mutually authenticating each other (Figure 4). 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 an 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.

4.2. Remote Memory Operations

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 Section 2.3). The host can therefore reliably obtain the memory pages of the normal world to enable the kinds of analyses described in Section 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 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

[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.

design tradeoff of minimizing the functionality of the secure world while allowing the rare remote write failure to happen.

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 value $r||\mathit{MemState}||\mathsf{HMAC}_{k_s}[r||\mathit{MemState}]$ 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 $\mathsf{VTok}[r_{checkin}]$ 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 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. Any attempts to undo the host's memory updates performed at check-in will thus be detected by the host.

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 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 [3]. 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. If the user does not choose this option, the device shuts down as usual, 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.

If the user 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.

We achieve this goal using a feature of the ARM TrustZone that provisions each device with a statistically-unique one-time programmable secret key, which we will refer to as $\mathsf{K}_{\mathsf{Dev}}$. $\mathsf{K}_{\mathsf{Dev}}$ is located in an on-SoC cryptographic accelerator, and accessible only to secure world software [3, §6.3.1]. $\mathsf{K}_{\mathsf{Dev}}$ cannot be read or changed outside the secure world, other bus masters or by the JTAG [27]. $\mathsf{K}_{\mathsf{Dev}}$ allows confidential data to be encrypted and bound to the device, and has previously been leveraged in other research [15, 42]. Note that $\mathsf{K}_{\mathsf{Dev}}$ is not the same as $\mathsf{PrivKeyG}$, 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 $\mathsf{B}_{\mathsf{REM}}$ to record that the device is REM-suspended. It stores the checkpoint and $\mathsf{B}_{\mathsf{REM}}$, together with an HMAC of these values under k_s on the device's persistent storage. The untrusted device user does not know $\mathsf{K}_{\mathsf{Dev}}$, and therefore cannot forge the encrypted value of k_s. The HMACs under k_s protect the integrity of the normal world checkpoint and $\mathsf{B}_{\mathsf{REM}}$.

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

5. The Vetting Service

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 4 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.

6. Implementation and 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.

We built a bare-metal 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) [2]. Excluding the cryptography library, our secure world consists of about 3,500 lines of C code, including about 250 lines of inline assembly.

Figure 5 shows the sizes of various components. We used PolarSSL's implementation of SHA1 and HMACs, RSA and X509 certificates. As shown in Figure 5, 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.

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

Figure 5: Sizes of components executing on the guest.

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 [37] 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 [9, 14, 37] based on the recursive snapshot traversal method outlined in Section 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.

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

Peripheral	Method Used (see Figure 2)	Bytes modified or added	Device used
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

Figure 6: Peripherals uninstalled using remote write operations to a guest device.

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 Section 4, they allow us to evaluate the ability to uninstall a variety of peripherals from a running guest device.

Figure 6 shows the set of peripherals that we uninstalled, the method used to uninstall the peripheral (from Section 3), the size of the write operation (*i.e.,* the number of bytes that we had to modify/introduce in the kernel), and whether the operation was performed on the i.MX53 or the Nexus. 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 Section 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.

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 Figure 6. 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.

Figure 7 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.

7. Related Work and Design Alternatives

TrustZone Support. A number of projects have used TrustZone to build novel security applications. TrustDump [47] is a TrustZone-based mechanism to reliably acquire memory pages from the normal world of a device (Android LiME [23] and similar tools [22, 46, 49] 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 [8] and SPROBES [21] 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 (Section 2.3).

TrustZone has also been used to improve the security of user applications. Microsoft's TLR [42] and Nokia's ObC [28] 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 [30] and securing mobile payments [39].

Enterprise Security. With the growing "bring your own device" (BYOD) trend, a number of projects have developed enterprise security solutions that enable multiple persona (*e.g.,* [6, 13, 24]) or enforce mandatory access control policies on smart devices (*e.g.,* [13, 24, 44, 50]). Prior work has also explored context-based access control and techniques for restricted space objects to push usage policies onto guest devices (*e.g.,* [16, 32, 34, 35, 40, 41]).

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 [24] 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 [4, 6, 17, 19]. In this approach, a trusted hypervisor on the guest device enforces isolation between VMs 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 [26, 41]. These techniques are able to do so because they

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 sitution 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.

Figure 7: Results of robustness experiments for user apps.

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 [45]. 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.

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 [7, 25]. This work has since been repurposed to perform kernel malware detection [38] and remote repair [12]. 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 [27], 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.

8. Concluding Remarks

This paper develops 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 paper 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 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 pursue these questions in follow-on research.

References (All URLs verified on August 12, 2015)

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

[2] PolarSSL: Straightforward, secure communication. https://polarssl.org.

[3] 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.

[4] 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.

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

[6] 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.

[7] The InfiniBand Trade Association. The InfiniBand^{TM} architecture specification. http://www.infinibandta.org.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

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

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

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

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

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

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] 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.

[33] 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.

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

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

[36] 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.

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

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

[39] Proxama. TapPoint® and Digital Enablement Platform (DEP), enabling mobile proximity commerce. http://www.proxama.com/platform.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

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

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

[47] 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.

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

[49] 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.

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