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

Saeed Mirzamohammadi, Justin A. Chen, Ardalan Amiri Sani, Sharad Mehrotra, Gene Tsudik
Department of Computer Science
University of California, Irvine
saeed@uci.edu, jachen1@uci.edu, ardalan@uci.edu, sharad@ics.uci.edu, gene.tsudik@uci.edu

ABSTRACT

Mobile and Internet-of-Things (IoT) devices, such as smartphones, tablets, wearables, smart home assistants (e.g., Google Home and Amazon Echo), and wall-mounted cameras, come equipped with various sensors, notably camera and microphone. These sensors can capture extremely sensitive and private information. There are several important scenarios where, for privacy reasons, a user might require assurance about the use (or non-use) of these sensors. For example, the owner of a home assistant might require assurance that the microphone on the device is not used during a given time of the day. Similarly, during a confidential meeting, the host needs assurance that attendees do not record any audio or video. Currently, there are no means to attain such assurance in modern mobile and IoT devices. To this end, this paper presents Ditio, a system approach for *auditing sensor activities*. Ditio records sensor activity logs that can be later inspected by an auditor and checked for compliance with a given policy. It is based on a *hybrid security monitor architecture* that leverages both ARM's virtualization hardware and TrustZone. Ditio includes an authentication protocol for establishing a logging session with a trusted server and a formally verified companion tool for log analysis. Ditio prototypes on ARM Juno development board and Nexus 5 smartphone show that it introduces negligible performance overhead for both the camera and microphone. However, it incurs up to 17% additional power consumption under heavy use for the Nexus 5 camera.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Privacy protections**; • **Computer systems organization → Sensors and actuators**;

KEYWORDS

Mobile and IoT devices, Sensors, Security and privacy, Operating systems

ACM Reference Format:
Saeed Mirzamohammadi, Justin A. Chen, Ardalan Amiri Sani, Sharad Mehrotra, Gene Tsudik. 2017. Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices. In *Proceedings of SenSys '17, Delft, Netherlands, November 6–8, 2017,* 14 pages.
https://doi.org/10.1145/3131672.3131688

1 INTRODUCTION

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

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

To address this issue, this paper presents Ditio, a system for *auditing* sensor activities in mobile and IoT devices, with a small Trusted Computing Base (TCB). Ditio records sensor activity logs; when needed, an auditor can check these logs for compliance with a given policy. Ditio is designed and built based on a *hybrid security monitor* that uses TrustZone [2, 13, 14, 61] and recently added virtualization hardware [17, 18] in recent ARM processors, which are available in modern mobile and IoT devices (§3). The hypervisor layer, supported by virtualization hardware, enables Ditio to efficiently record every access to registers of selected sensors without making any modifications to the operating system. TrustZone, on the other hand, enables sealing of logs to guarantee integrity, authenticity, and confidentiality.

Ditio is designed with practicality in mind. Its design is low-level, generic, and on-demand. Its low-level design allows it to be deployed and locked by the mobile and IoT device vendors (using the secure boot feature [13]) without modifications to the operating system. Its generic design allows it to support various sensor brands and different use-cases without additional engineering effort. It also allows Ditio to be easily ported to different mobile and IoT devices with diverse hardware configurations. Its on-demand nature means that logging can be turned on and off as needed. Hence, Ditio's runtime overhead, although not significant, can be fully avoided when sensor activity logging is not needed.

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

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

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

Ditio is related to our previous work on Viola [50], which provides formally verified sensor notifications. That is, it guarantees that a notification, such as LED light or vibration, is triggered if a privacy-sensitive sensor, such as camera or microphone, is used. Ditio and Viola provide complementary techniques for enforcing policies on the use of sensors. Viola performs runtime policy enforcement: it triggers a notification when the sensor is being used. Ditio enables auditing: it records accesses to sensors and allow future audits. These systems are not equally suitable for use cases involving sensor usage policy enforcement. For example, Viola cannot be reliably used for providing notifications to a third-party due to the lack of a reliable notification channel. On the other hand, Ditio is not suitable for scenarios where a delay in policy violation detection cannot be tolerated. It is, however, possible to use both systems together. For example, in the confidential meeting scenario discussed earlier, Viola can attempt to provide (but cannot guarantee the delivery of) a notification to the host when a sensor is being used. At the same time, Ditio can record logs in case a future audit is deemed necessary.

2 MOTIVATION

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

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

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

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

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

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

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

The second category of solutions are easier to implement but fail to provide any hard guarantees. For example, being aware of users' privacy concerns with home assistants, these devices provide a physical button for the user to turn the microphone off, while leaving the rest of the device on (e.g., its speakers). This can be used by the users to disable the microphone when privacy is needed [3]. Doing so might also change the LED color on the device for user's information [3]. However, a compromised device can easily bypass this measure (i.e., fake the LED color and keep the microphone on despite user's press of the button). As another example, the host of the confidential meeting might only verbally ask the attendees to turn their devices off or ask the hotel owner to turn off the wall-mounted cameras. However, he will not be able to detect any potential violations.

In contrast to these existng methods, Ditio provides an *easy-to-use, reactive, and rebust* solution. Ditio's auditing approach makes it easy for users to set up and use it and puts the burden on the device to provide adequate logs when needed. Moreover, Ditio provides strong guarantees due to its small TCB (§5) and its use of formal verification methods to enable the auditor to reliably analyze the recorded logs (§7).

3 HYBRID SECURITY MONITOR

Ditio's key component is a *hybrid security monitor* based on two important hardware features in modern ARM processors. **The first** is ARM TrustZone [2, 14, 61], which splits execution into two worlds: *normal world* hosting the main operating system and *secure world* hosting a secure runtime. **The second** is hardware support for virtualization, which creates a new privilege level in the normal world, called the hyp mode [30], in addition to existing kernel and user modes.

These hardware features are available on many new ARM processors. Examples are 32-bit processors, such as Cortex A7 and Cortex A15 [17], and 64-bit processors, such as Cortex A53 and Cortex A57 [18], which are used in most modern smartphones and tablets, such as Nexus 6P smartphone and Pixel C tablet (both using 4 Cortex A57 cores and 4 Cortex A53 cores in the big.LITTLE architecture). Moreover, these processors are used in modern high-end IoT devices and wearables as well. For example, Google Home leverages a dual-core ARM Cortext A7 processor [4]. Similarly, Samsung Gear 2 smartwatch leverages a dual-core ARM Cortext A7 processor in its Exynos 3250 SoC [10].

The main benefit of our monitor is its usage of complementary capabilities of both hardware features. It uses virtualization hardware for efficient monitoring of operating system's access to various registers using nested page tables, and uses TrustZone's secure world for sealing. Figure 1 overviews the hybrid security monitor. Communication between the hypervisor and secure world is attained through a protected channel implemented using shared memory and Secure Monitor Call (SMC) [13], which performs a context switch between normal and secure worlds.

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

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

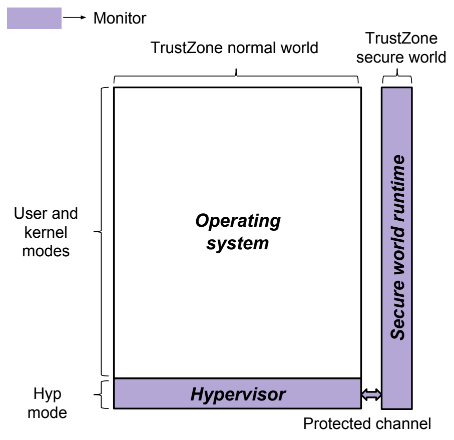


Figure 1: Hybrid security monitor overview.

4 OVERVIEW

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

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

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

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

4.1 Design and Workflow

Ditio consists of four components on the client: (*i*) a trusted sensor activity recorder implemented in the hypervisor, (*ii*) trusted authentication and sealing facilities in the TrustZone secure world runtime, (*iii*) an untrusted log store implemented in the normal world operating system kernel, and (*iv*) an untrusted configuration app in the normal world operating system user space. Ditio also includes an external trusted authentication server and a formally verified companion tool used by the auditor to identify violations of sensor activity policies. Figure 2 demonstrates Ditio's design.

Note that we explain the design of Ditio mainly in the context of the hybrid design (§3). The backward-compatible design is mostly similar except that the recorder and the authentication and sealing facilities are all in the operating system kernel communicating using

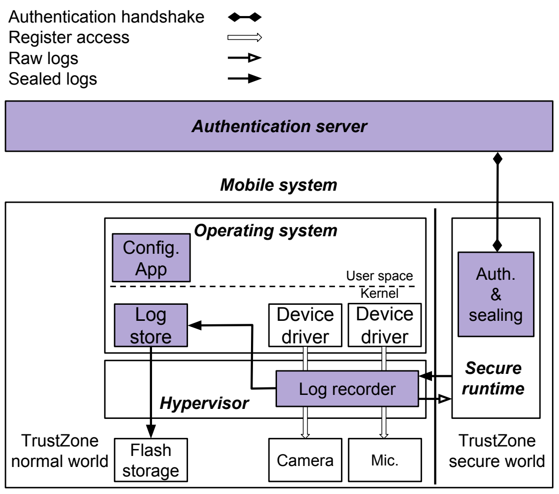


Figure 2: Ditio's components on the client (in dark).

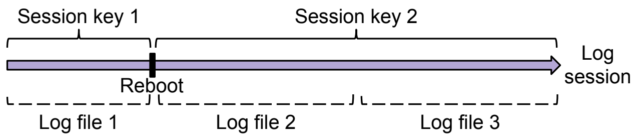


Figure 3: Log session example.

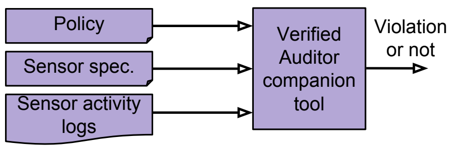


Figure 4: Ditio's audit process using the companion tool.

function calls. In the rest of the paper, we will highlight important differences when needed.

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

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

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

Sealing. Sealing includes encrypting the log buffer, and computing and appending its HMAC. Ditio encrypts the logs to protect its content since it can include sensitive information, such as when the camera was used. It appends the HMAC to protect integrity and authenticity of logs (§6.2).

Authentication. We use a symmetric key, i.e., *session key*, to seal the logs. The session key is shared between the client's secure world and a trusted authentication server through an authentication protocol performed in the beginning of the logging period and after every reboot in that period. The goal of this protocol is to: (*i*) inform the authentication server of the client identity, (*ii*) adjust the clock used to timestamp logs, and (*iii*) establish the session key. Note that communication between the secure world and authentication server is relayed by the normal world operating system. §6 describes this authentication protocol.

Log session. We refer to the logging period as the log session. A log session starts when the configuration app starts recording accesses to registers of sensors of interest and stops when the service stops recording on all registers. A log session can expand over multiple boots of the system. Figure 3 illustrates an example. Two session keys are used in the session, one for each boot. Moreover, there are three log files, the first with all the log entries in the first boot and the other two holding part of the entries in the second boot.

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

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

Auditing and the companion tool. Once logging is done, the owner of the client shares its logs with the auditor, if asked. The auditor then asks the authentication server to unseal the logs (§6.2). He then uses the companion tool to develop a policy and check the policy against the logs to detect potential violations. Note that this implies that the auditor and the authentication server will have access to the content of the logs and therefore the owner of the client must trust these two entities with the confidentiality of the logs. Figure 4 shows the audit process. In addition to the sensor activity logs and the query, the auditor also provides the sensor specification needed to analyze the logs (§7).

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

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

4.2 Threat Model

Ditio protects runtime integrity of the monitor against an attacker that compromises the operating system. This is because these components run in the hypervisor and the secure world, which are isolated from the operating system by hardware. Ditio also protects integrity, authenticity, and confidentiality of sensor activity logs as discussed. This is achieved by: (*i*) using a hardware-bound private key available in the secure world to authenticate the system to the authentication server and establish a shared session key, (*ii*) encrypting the log buffers and computing their HMAC using the session key, and (*iii*) leveraging a non-volatile counter in the secure world to associate log buffers of a single log session in a tamper-evident manner (§6).

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

As mentioned in §2, Ditio targets two categories of use cases: one where devices provide assurance for their owners and one that they do so for a third party. The latter (i.e., assurance for a third party) is vulnerable to physical attacks that cannot be protected by Ditio. Such attacks can come in two forms. First, the attacker can tamper with the client's hardware, e.g., by installing additional microphones or cameras. Second, the attacker can introduce an additional, hidden device. In the confidential meeting scenario, a malicious attendee can sneak in a wearable microphone. If needed, protecting against such attacks requires physical defenses, e.g., body search, and hence is beyond the scope of this paper. As related to physical attacks, we note that we assume the user of Ditio knows which devices to target for auditing. For example, in the meeting, the host may audit the smartphones, tablets, or smartwatches of attendees (those devices that are not hidden).

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

Ditio does not protect against side-channel attacks. This has two implications. First, Ditio cannot protect the confidentiality of logs against side-channel attacks. Second, an attacker can try to use other sensors as side-channels to record information of interest. For example, Michalevsky et al. showed that a gyroscope can be used to recognize speech, although the recorded signal frequency and recognition accuracy is low [49]. In this case, it is possible to use a different Ditio policy to also restrict the use of gyroscope in addition to microphone.

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

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

4.3 Trusted Computing Base

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

We believe that the hypervisor and secure world form a small and reliable TCB on the client. One concern is with security vulnerabilities of commodity hypervisors and secure world runtimes [21, 31].

Although we use commodity hypervisor and secure world runtime in our prototype (i.e., Xen and Open-TEE [8]), more secure hypervisors and runtimes can be used. This is especially the case for the hypervisor since it does not need to support many functionalities needed for running multiple virtual machines in the system. For example, it is possible to use a verified hypervisor or a microkernel [25, 39, 41, 43, 48].

The operating system kernel is not trusted in the hybrid design. However, in the backward-compatible design (§3), it hosts the log recorder and the authentication and sealing facilities and is thus trusted. The client hardware is trusted as well. The certification authority (CA), which issues the identity certificate for the client, and the authentication server are both also trusted.

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

5 RECORDING SENSOR ACTIVITY LOGS

To log sensor activity, Ditio records parameters of read and write accesses to sensor registers of interest. It also records other events needed for the auditing process, such as: (1) start and stop times of a session, (2) timestamps of power-on and power-off events in a session, and (3) the timestamp correction offset (computed vs. a reference time), as discussed in §6.1. In this section, we describe how the recorder works.

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

5.1 Untrusted Log Store

We designed the log store to be untrusted in order to minimize the TCB. More specifically, we use the existing operating system kernel and file system for implementing the log store to minimize the size and attack surface of the hypervisor. To do this, the hypervisor shares the logs over shared memory pages with the operating system. Note that the log buffers are encrypted and authenticated (using HMAC) in the secure world before they are shared with the operating system kernel (§4.1). Also note that the log store is untrusted in the backward-compatible design as well since the files on the file system are accessible to the user space.

5.2 Minimizing Recorder Latency

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

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

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

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

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

5.3 Configuring the Recorder

As discussed in §4.1, the configuration app requests the recorder to start or stop logging activities on a given register. To facilitate this, the hypervisor-based recorder provides a hypercall for the normal world operating system to request the start and end of logging on a given register.

The aforementioned hypercall requires the physical address of registers of interest. This leads us to consider how configuration

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

...

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

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

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

app learns the addresses of registers for various sensors. For this purpose, we use two resources: the *device tree file* to find out the range of physical addresses of all registers of the device, and the device driver or device specification to find out the offset of the registers of interest.

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

Note that in practice, Ditio does not need to monitor all the registers. For example, in case of Nexus 5 camera, logging only one register is enough to infer the on-off state of the device (§7). The information about which registers need to be monitored can be easily provided by the device vendors that deploy Ditio. In practice, as mentioned earlier, we identify the right registers by inspecting the sensor's device driver or the device specification. For example, it took us less than a day to identify the registers that we needed to monitor for the Nexus 5 camera by inspecting its device driver.

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

Trap-not-log: Protection of register accesses (i.e., forcing them to trap) happens at register page granularity. However, a register page contains many registers, not all of which might be of interest in the auditing phase. Therefore, Ditio avoids logging accesses to other registers in the same page that are not of interest. We implement *trap-not-log* to achieve this: accesses to all the registers in the page are trapped, yet only those of interest are logged. Note that all trapped register accesses are emulated regardless of whether they are logged or not. In §9, we show that trap-not-log, while simple to realize, is indeed important in reducing the overhead of Ditio.

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

An alternative is to remember monitored registers by storing them in secure storage of the secure world and re-apply the page table protection after reboot. We opted for the previous approach since it simplifies the design by not requiring a large amount of secure storage, which is indeed not trivial to implement [54].

6 SEALING THE LOGS

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

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

6.1 Authentication Protocol

At the start of each log session and after every reboot, the secure world performs an authentication handshake (Auth-Prot) with an authentication server (AuthSrv). As part of it, the client proves its identity to this server, synchronizes its wall clock with it, and establishes a symmetric key (i.e., a session key). In the context of Auth-Prot, we make the following assumptions. First, we assume that the AuthSrv is trusted. This can either be a trusted public entity or a private entity designated for Ditio. Second, we assume that the client has a certificate issued by a well-known and trusted certification authority (CA), e.g., the client vendor, such as Apple, Samsung, Google, and Cisco. This certificate securely binds the client's unique identity to a distinct public key. The public key corresponds to a hardware-bound private key, which is only accessible to the TrustZone secure world in the monitor (and not even the hypervisor). §8.1 describes how we use a hardware-unique key in TrustZone's secure world to implement a hardware-bound private key. In our backward-compatible design, we include this key in the operating system image.

The protocol works by exchanging two messages. We describe each message using notation similar to the one used by Needham et al. [52]. That is: $A \rightarrow B :\ C$ means that entity A sends message C to entity B.

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

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

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

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

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

Next, AuthSrv replies to M with a message that contains its current timestamp (T_{AuthSrv}) and an ID_{S_k}, which is simply an ID for S_k. Later on, M appends ID_{S_k} to the log file encrypted with S_k. In the audit phase (§6.2), AuthSrv uses ID_{S_k} to retrieve S_k. The message also includes an HMAC computed with S_k over T_{AuthSrv}, ID_{S_k} as well as over the entire previous message (MSG_1). HMAC guarantees integrity and origin authenticity of the message; it also (since MSG_1 includes N_M) authenticates AuthSrv to M. T_{AuthSrv} allows M to compute its wall clock offset compared to a reference time. We assume that AuthSrv can provide an accurate timestamp based on an agreed-upon time reference. Note that we do not consider the time taken to send and receive messages between M and AuthSrv. This might result in a clock skew. However, we expect it to be on the order of milliseconds and thus acceptable for anticipated use-cases. To defend against delay attacks, the secure world must terminate the handshake if a response is received after a delay threshold.

Finally, we note that the first message in our protocol has similarity to the well-known X.509 Authentication Procedure [40], but our second message is different.

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

6.2 Audit Phase

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

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

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

Log integrity: The appended HMAC also guarantees the integrity of each log file. However, unfortunately, merely protecting integrity of each log file is not adequate as the attacker can try to delete some log files and violate integrity of the log session. Therefore, we need to make sure that the session's log files are connected to each other in a tamper-evident manner, such that these attacks can be detected. We achieve this by using a *non-volatile counter* in the secure world [19]. The secure world sets the counter to one in the beginning of the log session and re-sets it to zero upon session termination. When preparing a log file, the secure world inserts the counter into the file and increments it. This way, the auditor can always verify that log files start at one and that all the subsequent log files are available. The last log file should contain an event that shows recording on all pages were terminated. In case of per-CPU log buffers discussed in §5.2, log files are tagged with counters in the order they are passed to the secure world.

Note that Ditio relies on some form of non-volatile storage in the secure world to maintain this counter value. As described in the TrustZone's specification [13], the secure world provides such a counter (although our development board does not fully implement this as described in §8.1 forcing us to emulate it). Moreover, TrustZone-based systems can provide secure storage support [5] but the implementation of that is challenging and missing from processors manufactured by all major SoC vendors [54]. Therefore, we do not leverage this storage to simplify the monitor.

7 FORMALLY VERIFIED COMPANION TOOL

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

As mentioned in §1, while the recorder itself enjoys a generic device-agnostic design, the content of the logs are specific to the sensors that are being recorded. Each sensor has a unique hardware interface consisting of several registers with different effects on the behavior of the sensor. Moreover, each register is often an

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

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

Figure 6: The implementation of a policy query in the high-level query language.

aggregation of multiple variables, hence requiring low-level bitwise operations for log analysis. Finally, the behavior of the sensor depends on other components as well, such as its peripheral bus. Auditor's attempt to analyze such low-level logs without additional help can cause misinterpretations of the logs, leading to errors.

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

Figure 6 shows an example of a policy query on the Nexus 5 camera written in this language. The language consists of two main programming constructs: *State* and *Query*. Using *State*, we specify the target state of the sensor that is of interest to the auditor. Using *Query*, we generate the query by passing the target state, the start time, and the end time. The first parameter, target state, specifies the state that the auditor is inspecting. In this case, he is looking to find instances of attempts to turn on the camera clock (which is turned on if the camera is turned on). The last two parameters define the time period, during which this query will look for policy violation.

We formally verified correctness of the translation of the checks from the high-level query to assembly. The formal proof provides an important advantage: it enables the auditor to prove correctness of its decision to others. To do this, the auditor can provide the high-level audit query, the low-level checker, and the proof of the correctness of the translation. The audit query is often simple and can be easily inspected by all parties (Figure 6). Moreover, if needed, one can check the proof and run the check against the logs to arrive at the same decision as the one by the auditor. This is mainly possible due to the small size of our companion backend: about 500 lines of code for the translator (in Coq) and an interface module to pass input to the generated checks.

To prove correctness of the checker code, we prove that if the sensor specification is correct, the checker that the backend generates is a correct assembly translation of the query provided by the auditor. We perform formal verification using Coq and its proof assistant [11]. The translator is fully implemented in Coq and proved correct using forward simulation [46, 47], similar to other systems [50, 59]. Specifically, our translator translates the code in query language to Cminor, which is an intermediate form of the formally verified C compiler, CompCert [44]. We provide a proof of soundness and completeness of this translation. Translation from Cminor to assembly code is then achieved by CompCert, which is already verified.

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

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

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

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

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

The translator requires the sensor specifications to generate the checks in the companion backend. As noted by others [55], I/O device specifications are increasingly available from vendors. Moreover, Ditio does not require full specifications of a sensor. If the policy is not concerned with a functionality of the device and is only concerned with the device being on and off, the specification can be a few tens of lines of code (§9.1).

Finally, some sensors are not directly mapped in the CPU address space. Instead, they are accessed through a peripheral bus, such as I^2C. In these cases, Ditio records accesses to the peripheral bus adapter registers. The companion then infers the register accesses of the device from the register accesses of the peripheral bus adapter, similar to the bus Interpreter module in [50]. Similar to the senors themselves, we expect the specification of the bus to be available from their vendors. However, if not, it is possible for the system designer to develop the required partial specification. To demonstrate this, we have developed the specification of the I^2C bus for the Juno board, which took one of the authors a few days to do.

8 IMPLEMENTATION

We implement the full Ditio prototype on ARM Juno r0 development board. The board incorporates the ARM big.LITTLE architecture, with the big cluster of Cortex-A57 CPUs and the LITTLE cluster of Cortex-A53 CPUs, both of which use the ARMv8-A 64-bit instruction set. To the best of our knowledge, ARM Juno development boards (r0, r1, and r2) are the only devices that allow programming of the hypervisor and the TrustZone secure world to non-vendors. Note that we run the operating system in Xen's Dom0 in our prototype. Xen provides a command interface for Dom0 to manage the system, e.g., launch and kill new virtual machines. Since such functionality is not needed in Ditio, this provileged interface of Dom0 must be disabled, although we have not done so in our prototype. We also implement a backward-compatible version of Ditio on a Nexus 5 smartphone since we cannot access the hypervisor and TrustZone's secure world on commodity devices (§3).

We support and test different sensors in our prototypes. On the Juno board, we test a USB-based camera (Logitech C270 HD 720p). On the Nexus 5, we test the memory-mapped backfacing camera and the SLIMbus-based microphone. As mentioned in §5.3, we deploy trap-not-log for the Nexus 5 camera. For the other two devices in our prototype, we simply log all the register accesses to their corresponding peripheral buses. While we only support camera and microphone in our current prototype (since they are some of the most privacy-sensitive sensors), we believe that Ditio can be easily applied to other sensors as well, e.g., GPS.

In our Juno board prototype, we use Xen for the hypervisor (version 4.8), OpenEmbedded operating system [9] (version 16.04) running on Linux kernel (version 4.6) for the normal world operating system, and Open-TEE [8] (version 16.04) secure operating system for our secure world runtime. We use the mbed TLS library [6] to perform the Ditio's cryptographic functions in the secure world (for the hybrid design) or in the kernel (for the backward-compatible design). The library provides support for AES encryption, RSA encryption and signing, and HMAC. We use the same library in the authentication server. In our Nexus 5 prototype, we use the Android operating system (CyanogenMod version 12.1) running on the Linux kernel (version 3.4).

8.1 Juno Board's Specifics

Three issues about the Juno development board are noteworthy.

First, TrustZone specification [13] envisions a "statistically unique secret key" on the SoC. Juno development board's implementation of this key is a 128-bit hardware unique key stored in "One Time Programmable (OTP) or eFuse memory" [19]. As mentioned in §6.1, Ditio uses a hardware-bound private key for digital signature. We use a 2048-bit RSA key hard-coded in the secure world image. For complete secrecy, the RSA key must be encrypted with the hardware unique key and then hard-coded in the image. We note that while the Juno development board supports the hardware unique key, some other implementations of TrustZone do not. Ditio relies on the availability of this key to implement its hardware-bound key.

Second, the session key in Ditio is a 128-bit AES key generated by the client. The Juno development board provides a trusted entropy source to generate the session key [19].

Third, the TrustZone specification provisions a 32-bit non-volatile counter [13], needed in Ditio to connect the log files in a tamper-evident manner (§6.2). This counter can be used to number and chain more than 4 billion log files. We believe that this number is large enough and that we will not face counter overflow problems in any practical scenarios. In the unlikely case that the counter overflows, the secure world can finish the logging session and starts a new one. The user will then need to provide the logs for both sessions to the auditor.

Following the specification, the Juno board does provision a non-volatile counter in the secure world, however, according to the board documentation [19] and based on our experiments, the value of the counter is always fixed at 31. Therefore, we emulate this counter in our prototype. Moreover, as mentioned in §4.1, Ditio can use the secure storage (i.e., storage space protected by the secure world) for maintaining the counter, an option we have avoided to simplify the monitor (§6.2).

9 EVALUATION

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

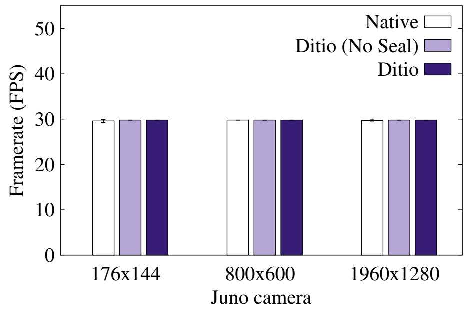
9.1 Use Cases

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

The most important question to answer about this experiment is the ease of use of Ditio. Our experience was that the most time-consuming part of using Ditio was developing the sensor interface specification. Fortunately, the specification is partial; that is, it only captures the registers of interest to the event that needs to be detected. As a result, the specification for the camera on Nexus 5 is 13 lines of Coq code. It took one of the authors only a few days to implement this. Note that we expect these specifications to be provided by sensor vendors (§7). This will then significantly reduce the engineering effort needed to deploy Ditio.

9.2 Sensor Performance

We measure the performance of different sensors when being recorded by Ditio and show that the performance overhead is negligible. For every sensor, we measure the performance of the sensor when used natively, and when used while being recorded. We show the Ditio's overhead with and without sealing in the secure world (the latter marked as "No Seal" in the figures), to quantify the effect of sealing on performance. Moreover, for Nexus 5 camera experiments, we also show the results with trap-not-log disabled to demonstrate



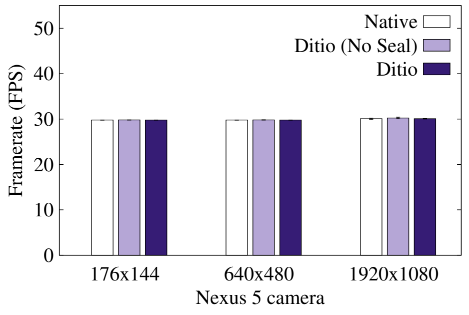


Figure 7: Juno board and Nexus 5 camera performance.

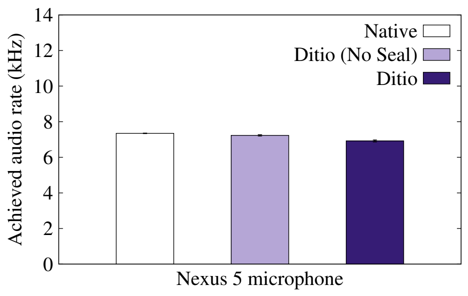


Figure 8: Nexus 5 microphone performance.

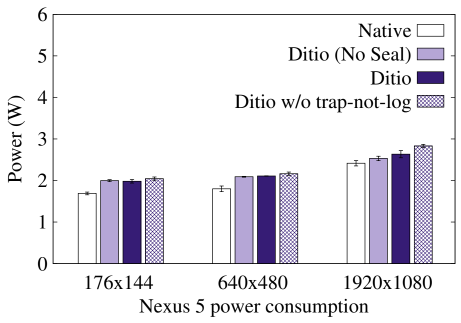


Figure 9: Power consumption of using the Nexus 5 camera.

the effectiveness of this technique (§5.3). Every reported performance number is an average over three runs. For every average number, we also include the standard deviation using error bars in the figures. Note that we do not run any specific applications in the system while evaluating Ditio nor do we pin any of our software components to any cores in the device. In any experiment, we reboot the system and launch our applications for testing such as camera or a recorder application.

Camera performance We measure the performance of the camera by measuring the rate at which it can capture frames, i.e., framerate. We configure the camera to produce the frames at its highest framerate possible (i.e., 30), run the camera for about 1 minute, and measure the rate. We ignore the first 50 frames to remove the effect of camera initialization on the results. Figure 7 shows the results for the camera on both the Juno board and Nexus 5 smartphone for different resolutions. We use the IP Webcam [1] application on Nexus 5 and a simple frame-capture program on the Juno board. The results demonstrate that Ditio does not add any noticeable overhead to the performance of the camera.

Microphone performance We measure the performance of the microphone on Nexus 5. Note that we have not implemented support for microphone on our Juno board (§8).

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

Figure 8 shows the results. The results demonstrate that Ditio's affect on the microphone performance is small.

9.3 Power Consumption

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

We measure the power consumption for Nexus 5 experiments using a Monsoon power monitor [7]. We focus on camera since it stresses Ditio. Microphone incurs negligible overhead (within the margin of error in our measurement setup). Figures 9 shows the results. It shows that logging by Ditio (when camera is actively used) increases the power consumption of the device by at most 17% (for the 176×144 resolution).

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

9.4 Other Results

Log size: We measure the log size generated in each of the experiments in §9.2. Our results show the log size to be about 10 MB/min for the camera on the Juno board, and 16 kB/min and 8 kB/min for camera and microphone on Nexus 5.

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

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

10 RELATED WORK

10.1 I/O Restriction & Auditing

Brasser et al. built a system to regulate the use of mobile I/O devices (including sensors) in restricted spaces [24]. In their solution, the owner of the restricted space, i.e., the host, disables the targeted I/O device in the guest's mobile system for the period of the visit (i.e., from a check-in time to a check-out time). To do this, the host performs remote memory writes to the kernel memory of the guest's mobile device to nullify the device driver entry points or to install dummy device drivers. Unfortunately, this solution has two important limitations: First, it requires the guest to allow the host to perform memory operations on the kernel memory of the mobile system, which can be used by a malicious host to mount serious attacks on the client. The system addresses the problem by leveraging a vetting service, which can monitor and authorize the requested memory operations by the host. However, implementation of a vetting service that can protect against all possible attacks is challenging. This is because having access to the kernel memory provides a significantly large attack surface for the host. Second, this is a semi-preventive solution, since a re-boot resets the checks on the mobile device, thus enabling the use of I/O devices. Although a reboot can be later detected by the host (similar to our reactive approach), I/O usage is not fully prevented. Moreover, after reboot, nothing can be said about the activity of I/O devices.

In contrast to this work, Ditio provides an auditing framework for sensor activities and solves the limitations mentioned above. First, Ditio minimizes the attack surface on the client by only requiring it to run a trusted monitor, which is inspected and deployed by the device vendor. Second, rather than attempting to provide a semi-preventive solution, Ditio provides a reactive solution, which is also robust against reboots.

Wilson et al. [60] designed TLS-Rotate and Release (TLS-RaR), a solution for auditing the data transmitted from an IoT device to its server. The solution is realized at the network layer and is applicable to IoT devices that use TLS for securing their communication. For these devices, TLS-RaR records all the network communication at a proxy. It then asks the device to release the key used for the TLS session. The device performs a TLS key update (to ensure forward integrity) before releasing the key. There are two important differences between TLS-RaR and Ditio. First, unlike TLS-RaR, which is only applicable to IoT devices connected to the network through a proxy, Ditio supports both mobile and IoT devices regardless of their connectivity. Second, TLS-RaR audits the data transmitted by IoT devices whereas Ditio audits the state of sensors in the device. Therefore, TLS-RaR cannot unmistakenly determine the state of the sensor in the device. The sensor might be on recording and storing data without transmitting them. TLS-RaR can detect the transmission of data in the future but puts a burden on the user to analyze all the transmitted data and reason about them. These differences make Ditio and TLS-RaR suitable for different use cases. Ditio is best used when guarantees are needed about the use or non-use of the sensors in mobile and IoT devices at a given time in the past. TLS-RaR is best used when the actual transmitted data (through a proxy) by an IoT device needs to be viewed by the user regardless of when they were captured.

10.2 Trusted Sensors

Several systems provide support for trusted readings from sensors of mobile systems. Liu et al. [45, 57] use TrustZone to implement trusted sensors. Their solution has two main aspects: first, they execute the software stack needed for generating the sensor data in the secure world in order to protect it from the rest of the system, and second, they sign the sensor data to prove its authenticity.

Gilbert et al. [37, 38] provide a trusted sensor framework, where they use a trusted hardware, e.g., Mobile Trusted Module (MTM) [12], to provide signed statements about the released sensor readings, which can be transformed from the raw readings for better privacy.

This line of work is similar to ours in that Ditio provides authenticated information (i.e., logs) about sensors as well. However, Ditio's logs contain register-level information rather than high-level sensor data. Ditio's logs are best suited for determining the state of the sensor and not for reasoning about the data that it produced. In that sense, our work is complementary to this line of work.

10.3 Recording systems

Many systems log events in the operating system for various applications, such as replay and provenance [23, 51, 53]. In [23], Bates et al. present Linux Provenance Modules (LPM), which collects and records provenance information in the Linux kernel. LPM guarantees the integrity and authenticity of the recorded information. Similarly, Ditio collects and records sensor activity information and guarantees its integrity and authenticity. In contrast to LPM, which is implemented in the kernel, Ditio's main design is implemented in the hypervisor and hence has a smaller TCB. Moreover, unlike LPM, which is implemented in an x86 machine and uses a TPM as its root of trust, Ditio is implemented for mobile systems and uses ARM TrustZone's secure world as its root of trust.

10.4 TrustZone and Hypervisor-based Systems

Several systems use ARM TrustZone and hypervisor for security applications. TrustDump [58] uses ARM TrustZone to record the memory and register space of the normal world operating system when it crashes or is compromised. They use a secure interrupt to switch to the secure world. However, unlike Ditio, they do not log operating system events, such as register accesses. They also do not use the hypervisor.

TrustZone-based Real-time Kernel Protection (TZ-RKP) [20] used in Samsung KNOX [15] uses ARM TrustZone to implement memory protection solutions for smartphones. In TZ-RKP, the kernel is deprivileged and not allowed to execute certain control instructions, e.g., write to the TTBR register, which holds the address of the process page tables. Building on this, the secure world forces the kernel to ask the secure world to emulate these security critical instructions, allowing it to inspect them. Note that an approach like TZ-RKP is not ideal for monitoring sensor activity. This is because doing so requires the secure world to assign the sensor to the secure world. With such an approach, the secure world either has to host the device driver, which bloats the TCB, or has to incur a context switch per register access, which is costly. In addition, the operating system needs modifications too. In contrast, by leveraging both the hypervisor and the secure world, Ditio provides a solution for which no changes are needed to the operating system and the overhead of monitoring the writes to the registers is also significantly lower.

Similarly, in SPROBES [36], Ge et al. intercept certain events in the normal world operating system to trap in the secure world in order to perform introspection of the operating system. They do so by rewriting certain instructions in the operating system image to instead make an SMC call, which switches from the normal world to the secure world. In contrast, in Ditio, we use the Stage-2 page tables in the hypervisor to cause a trap for accesses to registers of interest to us, requiring no operating system modifications.

SchrodinText [16] uses TrustZone secure world and the normal world hypervisor to protect the textual content of applications on the display. It uses the hypervisor for controlling access to the framebuffer and uses the secure world for decrypting the ciphertext sent from the application server. In contrast, Ditio uses these hardware features to collect and seal sensor usage logs.

AdAttester [29] provides safety and attestation guarantees for mobile ads using TrustZone. It splits the graphics and input stacks and moves parts of them to the secure world, hence increasing the TCB size. Unlike AdAttest, Ditio keeps the TCB small. Moreover, it focuses on sensor activity and not mobile ads.

fTPM provides a software-based implementation of Trusted Platform Module (TPM) for ARM TrustZone [54]. Ditio can use fTPM in its monitor instead of the Open-TEE runtime.

TrustZone has also been used for secure credentials [42], secure facial authentication [62], and safe execution of applications (by offloading part of the application to the secure world) [56].

10.5 System verification

In Ditio, we use formal verification to prove correctness of the checkers we use for analyzing sensor activity logs. Many other systems have used verification tools for building trustworthy systems as well. Examples are a formally verified in-kernel interpreter (Jitk [59]), a certified crash-safe file system (FSCQ [26]), a verified C compiler (CompCert [44]), and verified microkernels and hypervisors (SeL4 [41] and others [25, 39, 43, 48]).

The closest work to Ditio is our previous work on Viola [50], which provides formally verified sensor notification invariant checks guaranteeing that a notification, such as LED light or vibration, is triggered if a privacy-sensitive sensor, such as camera or microphone, is used. Viola injects these invariant checks in the hypervisor in a mobile system. Viola's invariant checks operate at the register access level, similar to Ditio that records sensor activity at this layer. However, unlike Viola that tries to enforce these invariants at runtime, Ditio records sensor activity logs and audits them afterwards to find violations. Auditing has an important benefit: it can provide an undeniable evidence, which is useful in order to hold the violating parties accountable. Moreover, Ditio leverages ARM TrustZone for authentication and sealing in the monitor, in contrast to Viola, which only utilizes the virtualization hardware.

10.6 Others

Ditio isolates and protects its recorder. Hence it is related to existing sandboxing systems. Examples are SKEE [21] that creates an isolated environment in the operating system thus not relying on security hardware features. Ditio is also related to virtual machine introspection solutions (VMI) that use the hypervisor to monitor the events in the operating system, such as [28, 32, 34, 35].

Ditio allows a third-party to check the compliance of mobile and IoT devices with given policies. The policies in Ditio are simple: they regulate the use of sensors in a given period of time. Existing work has explored more sophisticated policies for data protection, resource usage control, and application behavior specification in mobile devices leveraging a richer set of contextual information (e.g., location) than Ditio does (only time) [22, 27]. Ditio's novelty is in how it records and then analyzes trustworthy logs of sensor activity.

Candid interaction [33] provides awareness of one's mobile and wearable activities to others in the vicinity using various techniques, such as augmented reality, for better social acceptable. Similarly, in some of its use cases, Ditio provides information about the sensor usage in mobile and wearables to a third party. However, Ditio focuses on privacy and shares this information with a user-approved third party.

11 CONCLUSIONS

We presented Ditio, a system solution for auditing sensor activities in mobile and IoT devices. Ditio enables important use cases that require auditing by device owners or third parties. We presented the design of Ditio based on a hybrid security monitor that leverages both ARM TrustZone and virtualization hardware. We also presented an authentication protocol that allows the auditor to verify the integrity and authenticity of the logs. Moreover, we discussed a formally verified companion tool that allows the auditor to analyze the device-specific logs with ease and without errors. We presented a prototype of Ditio based on ARM Juno development board, the only device that allows programming of both the hypervisor and TrustZone secure world by non-vendors, as well as a backward-compatible prototype on Nexus 5 smartphone. Our experiments showed that Ditio does not add noticeable sensor performance overhead, but that it incurs up to 17% additional power consumption.

ACKNOWLEDGMENTS

The work was supported in part by NSF Award #1617513. The authors thank the shepherd, Anthony Rowe, and the reviewers.

REFERENCES

[1] Android IP Webcam application. https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en.
[2] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.
[3] Disabling the microphone on Google Home (item 11). https://www.cnet.com/how-to/google-home-tips-and-tricks/.
[4] Google Home. https://www.ifixit.com/Teardown/Google+Home+Teardown/72684.
[5] Introduction to Trusted Execution Environments (TEE). http://sec.cs.ucl.ac.uk/users/smurdoch/talks/rhul14tee.pdf
[6] mbed TLS. https://tls.mbed.org/.
[7] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.
[8] Open-TEE. https://open-tee.github.io/.
[9] OpenEmbedded. http://www.openembedded.org/.
[10] Samsung Gear 2 uses Exynos 3250 SoC. https://www.sammobile.com/2014/04/09/sammobile-confirms-new-exynos-3250-soc-powers-gear-2-gear-fit-uses-cortex-m4-chip/.
[11] The Coq Proof Assistant. https://coq.inria.fr/.
[12] Trusted computing group mobile specifications. http://www.trustedcomputinggroup.org/work-groups/mobile/
[13] 2004. ARM Security Technology, Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
[14] 2004. TrustZone: Integrated Hardware and Software Security: Enabling Trusted Computing in Embedded Systems. In *ARM White Paper*.
[15] 2013. An overview of Samsung KNOX. In *Samsung White Paper*.
[16] A. Amiri Sani. 2017. SchrodinText: Strong Protection of Sensitive Textual Content of Mobile Applications. In *Proc. ACM MobiSys*.
[17] ARM. 2013. ARM Cortex-A15 MPCore Processor Technical Reference Manual, Revision: r4p0. *ARM DDI* 0438I (ID062913).
[18] ARM. 2013, 2014, 2016. ARM Cortex-A57 MPCore Processor Technical Reference Manual, Revision: r1p3. *ARM DDI* 0488H.
[19] ARM. 2014. Juno ARM Development Platform SoC, Revision r0p0, Technical Overview. https://static.docs.arm.com/dto0038/a/DTO0038A.pdf, *ARM DTO* 0038A (ID040516).
[20] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM Trustzone Secure World. In *Proc. ACM CCS*.
[21] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. 2016. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proc. ACM MobiSys*.
[22] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen. 2010. Context-Aware Usage Control for Android. In *Proc. Int. ICST Conference on Security and Privacy in Communication Systems (SecureComm)*.
[23] A. Bates, D. Tian, K. R.B. Butler, and T. Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proc. USENIX Security Symposium*.
[24] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A. Sadeghi. 2016. Regulating ARM TrustZone Devices in Restricted Spaces. In *Proc. ACM MobiSys*.
[25] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proc. ACM PLDI*.
[26] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proc. ACM SOSP*.
[27] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. 2012. CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android. In *Proc. IEEE Transactions on Information Forensics and Security*.
[28] L. P. Cox and P. M. Chen. 2007. Pocket Hypervisors: Opportunities and Challenges. In *Proc. IEEE/ACM HotMobile*.
[29] J. Crussell, R. Stevens, and H. Chen. 2014. Madfraud: Investigating Ad Fraud in Android Applications. In *Proc. ACM MobiSys*.
[30] C. Dall and J. Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proc. ACM ASPLOS*.
[31] National Vulnerability Database. Vulnerability Summary for CVE-2015-6639. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6639
[32] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proc. IEEE Security and Privacy (S&P)*.
[33] B. Ens, T. Grossman, F. Anderson, J. Matejka, and G. Fitzmaurice. Candid Interaction: Revealing Hidden Mobile and Wearable Computing Activities. In *Proc. ACM UIST*.
[34] Y. Fu and Z. Lin. 2012. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proc. IEEE Security and Privacy (S&P)*.
[35] T. Garfinkel and M. Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Internet Society NDSS*.
[36] X. Ge, H. Vijayakumar, and T. Jaeger. 2014. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Proc. IEEE Mobile Security Technologies Workshop (MoST)*.
[37] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. 2010. Toward Trustworthy Mobile Sensing. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*.
[38] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. 2011. YouProve: Authenticity and Fidelity in Mobile Sensing. In *Proc. ACM SenSys*.
[39] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S. Weng, H. Zhang, and Y. Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proc. ACM POPL*.
[40] A. Kahate. 2013. *Cryptography and Network Security*. Tata McGraw-Hill Education.
[41] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SOSP*.
[42] K. Kostiainen, J. Ekberg, N. Asokan, and A. Rantala. 2009. On-board Credentials with Open Provisioning. In *Proc. ACM International Symposium on Information, Computer, and Communications Security (ASIACCS)*.
[43] D. Leinenbach and T. Santen. 2009. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. International Symposium on Formal Methods (FM)*. Springer.
[44] X. Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM*.
[45] H. Liu, S. Saroiu, A. Wolman, and H. Raj. 2012. Software Abstractions for Trusted Sensors. In *Proc. ACM MobiSys*.
[46] N. Lynch and F. Vaandrager. 1995. Forward and Backward Simulations Part I: Untimed Systems. *Information and Computation*.
[47] N. Lynch and F. Vaandrager. 1996. Forward and Backward Simulations Part II: Timing-Based Systems. *Information and Computation*.
[48] M. McCoyd, R. B. Krug, D. Goel, M. Dahlin, and W. Young. 2013. Building a Hypervisor on a Formally Verifiable Protection Layer. In *Proc. IEEE Hawaii International Conference on System Sciences (HICSS)*.
[49] Y. Michalevsky, D. Boneh, and G. Nakibly. 2014. Gyrophone: Recognizing Speech from Gyroscope Signals. In *Proc. USENIX Security*.
[50] S. Mirzamohammadi and A. Amiri Sani. 2016. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. In *Proc. ACM MobiSys*.
[51] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. 2006. Provenance-Aware Storage Systems. In *USENIX ATC*.
[52] R. M. Needham and M. D. Schroeder. 1978. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*.
[53] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. 2012. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proc. ACM Annual Computer Security Applications Conference (ACSAC)*.
[54] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. 2016. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16), Austin, TX*.
[55] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. 2014. User-Guided Device Driver Synthesis. In *Proc. USENIX OSDI*.
[56] N. Santos, H. Raj, S. Saroiu, and A. Wolman. 2014. Using ARM TrustZone to Build a Trusted Language Runtime for Mobile Applications. In *Proc. ACM ASPLOS*.
[57] S. Saroiu and A. Wolman. 2010. I Am a Sensor, and I Approve This Message. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*.
[58] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. 2014. TrustDump: Reliable Memory Acquisition on Smartphones. In *Proc. European Symposium on Research in Computer Security (ESORICS)*.
[59] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. 2014. Jitk: a Trustworthy In-Kernel Interpreter Infrastructure. In *Proc. USENIX OSDI*.
[60] J. Wilson, R. S. Wahby, H. Corrigan-Gibbs, D. Boneh, P. Levis, and K. Winstein. 2017. Trust but Verify: Auditing Secure Internet of Things Devices. In *Proc. ACM MobiSys*.
[61] J. Winter. 2008. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *Proc. ACM Workshop on Scalable Trusted Computing (STC)*.
[62] Dongli Zhang. 2014. TrustFA: TrustZone-Assisted Facial Authentication on Smartphone. *Technical Report*.