Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems

Saeed Mirzamohammadi, Ardalan Amiri Sani

University of California, Irvine
saeed@uci.edu, ardalan@uci.edu

Abstract

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

To address these concerns, we believe that mobile systems must be equipped with *trustworthy sensor notifications*, which use indicators such as LED to inform the user unconditionally when the sensors are on. We present *Viola*, our design and implementation of trustworthy sensor notifications, in which we leverage two novel solutions. First, we deploy a runtime monitor in low-level system software, e.g., *in the operating system kernel or in the hypervisor*. The monitor intercepts writes to the registers of sensors and indicators, evaluates them against checks on sensor notification invariants, and rejects those that fail the checks. Second, we use *formal verification methods* to prove the functional correctness of the compilation of our invariant checks from a high-level language.

We demonstrate the effectiveness of Viola on different mobile systems, such as Nexus 5, Galaxy Nexus, and ODROID XU4, and for various sensors and indicators, such as camera, microphone, LED, and vibrator. We demonstrate that Viola incurs almost no overhead to the sensor's performance and incurs only small power consumption overhead.

Keywords

Mobile systems; Sensors; Notifications; Indicators; Invariants; Virtualization; Verification

1. INTRODUCTION

Modern mobile systems such as smartphones, tablets, and wearables have one important property in common: they contain a plethora of sensors. A smartphone today, for example, contains tens of sensors including camera, microphone, GPS, accelerometer, and fingerprint scanner. Moreover, being mobile, these systems are with the user at all times. Such a usage model exposes mobile sensors to extremely private information about the user, including daily conversations, photos, videos, and visited locations, all of which is considered to be sensitive by some or most users according to a study by Muslukhov et al. [46].

Such a powerful sensing capability raises important privacy concerns for users. These concerns are reinforced given the recent incidents where women were spied on through their webcams [7, 8] and where users were spied on through their smartphone microphones and cameras [6]. In fact, in Android, the camera and microphone were shown to be hacked and controlled remotely using a commercialized trojan [4].

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

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

MobiSys'16, June 25 - 30, 2016, Singapore.

ISBN 978-1-4503-4269-8/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2906388.2906391

Figure 1: LED notification for microphone using Viola. The LED blinks even if the application is in the background or if the display is off. Malicious attempts to break this notification will be blocked.

in the background and with the display off. Second, laptop webcams often notify the user with a built-in LED. These notifications are implemented by the webcams themselves and do not apply to other sensors. Moreover, as Brocker et al. [18] showed, these notifications can be circumvented in some MacBook laptops and iMac desktops by rewriting the webcam firmware, demonstrating the difficulty of implementing trustworthy sensor notifications.

We present *Viola*, a system solution for providing trustworthy sensor notifications in mobile systems. Our fundamental observation is that it is possible to formulate a sensor notification as a logical *invariant* on the hardware states of the sensor and indicator. For example, for an LED notification for microphone, the invariant is `microphone recording` $\rightarrow$ `LED blinks`, where $\rightarrow$ depicts logical implication. Viola employs runtime invariant checks in the system to guarantee that this invariant is never violated even if the system is compromised by an attacker. That is, the check does not allow the microphone to start recording unless the LED is blinking and does not allow the LED to be turned off while the microphone is recording. Figure 1 shows Viola in action. See a video demo of Viola in [12].

We answer two important questions about the design of the invariant checks.

Q1. Where should the runtime invariant checks be inserted? In Viola, we insert the checks in the low-level system software, e.g., in the operating system kernel or in the hypervisor. Since sensor notifications are mostly about constraints on the hardware states of the sensors and the indicators, it is possible to check the notification' invariant in the low-level system software irrespective of the state of higher level software. At these low-level layers, we intercept the writes to the registers of sensors and indicators and pass them to the invariant checks. We allow the successful execution of a register write only if it successfully passes all the checks.

Inserting the checks in the low-level system software enhances the trustworthiness of Viola as it reduces the size of the Trusted Computing Base (TCB). In this design, bugs in the device drivers and I/O services, which are very common [5,17,23,31,48], will not undermine the sensor notification invariants. Moreover, this design protects the integrity of the notification invariant against powerful malware including those with root or kernel privileges. The two layers mentioned above, i.e., kernel or hypervisor, provide a trade-off between trustworthiness, support for different operating systems, support for legacy systems, and performance, as will be explained in §4.

Q2. How can system designers develop provably correct invariant checks? Writing error-free invariant checks that operate on the parameters of writes to registers is fairly complicated given the complex hardware interface of many I/O devices, their peripheral buses, and the components they rely on, such as power supplies and clock sources in the System-on-a-Chip (SoC).

We tackle this challenge using an invariant language and its verified compiler. Viola's invariant language enables the developer to mainly focus on the invariant logic using a high-level and intuitive syntax. A verified compiler, which we build and verify using the Coq language and its proof assistant [11], guarantees that the generated invariant checks preserve the semantics of Viola's invariant language. The compiler uses device specifications for inferring device state transitions as a result of register writes.

We present an implementation of Viola that supports different sensors and indicators, such as camera, microphone, LED, and vibrator, on two smartphones, Nexus 5 and Galaxy Nexus. While hardware support for virtualization is increasingly available on mobile systems [13], commercial mobile systems either do not leverage this hardware support or, if they do, they do not provide open access support for programming the hypervisor. Therefore, to demonstrate the feasibility of using the hypervisor layer, we implement Viola on the ODROID XU4 development board as well.

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

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

Also note that Viola requires modifications to various parts of the system software and hence is not easily deployable by ordinary users on their mobile systems, e.g., by simply installing an application. We mainly envision Viola to be adopted by mobile system vendors, such as Samsung and LG. However, we believe that expert users can also benefit from Viola especially since Viola's verified compiler makes it easy for them to develop the required invariant checks.

In summary, we make the following contributions.

- We present Viola, a system solution for enabling trustworthy sensor notifications on mobile systems that inserts invariant checks in the low-level system software, such as the kernel or the hypervisor.
- We use formal verification and provide machine-checked proofs to guarantee the functional correctness of the compilation of invariant checks used in Viola from a high-level language. To the best of our knowledge, this is the first work to provide formal guarantees on the behavior of I/O devices (e.g., the relationship between the states of sensors and indicators).

2. SENSOR NOTIFICATIONS

In this section, we provide background information for sensor notifications including different types of indicators that can be used and different notification guarantees.

2.1 Indicators

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

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

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

Display. Display is an effective channel to convey notifications to the user. Indeed, in [49], authors demonstrate that showing a glyph on the display is more effective than an LED in capturing the user's attention. However, not only these notifications require the display to be in user's sight, they also require the display to be on, which is power hungry.

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

Also, note that our goal in this paper is to build a trustworthy framework for notifications. Determining the best indicator for each sensor requires a user study, similar to the study in [49], which is orthogonal to this work and is part of our future work.

2.2 Notification Guarantees

Causal guarantees. We envision two types of notifications in terms of causal guarantees provided to the user. *One-way* notifications guarantee that the indicator is on, e.g., the LED is illuminated, when the sensor is on. However, they do not provide any guarantees on the state of the indicator when the sensor is off. The logical invariant for one-way notifications can be expressed as `sensor in target state` $\rightarrow$ `indicator in target state`. For a simple uni-color LED, the target state can be as simple as LED being illuminated. Or for an RGB LED with built-in blinking capabilities and varying illumination levels [10], the target state may be that the LED blinks in a red color with a given frequency and illumination level. In the rest of the paper, we use the terms "in target state" and "on" interchangeably.

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

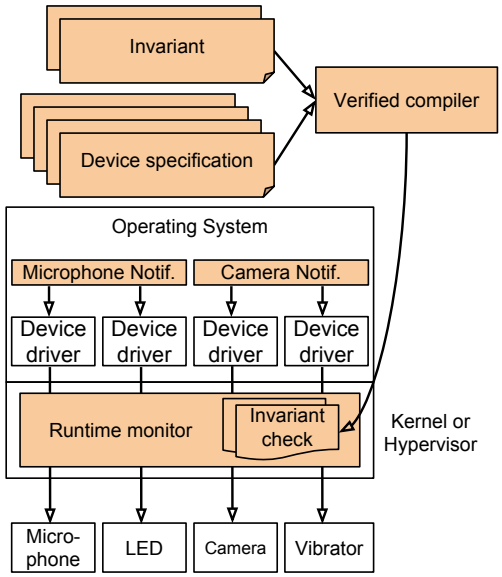

Figure 2: Viola's design. The darker components belong to Viola.

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

In this paper, we demonstrate the use of LED and vibrator as indicators while supporting one-way and continuous notifications. §8 discusses the challenges of providing formal guarantees for the use of speaker and display as indicators, and for two-way and temporary notifications. It further discusses our plans to overcome these challenges in the future.

3. OVERVIEW

Figure 2 illustrates the design of Viola. There are three main components. The first component is the implementation of sensor notification, which leverages the device driver (either directly or indirectly through user space I/O service API) to turn on the indicator before the sensor starts recording and to turn the indicator off after the sensor stops. This implementation is not trusted and hence errors in it will not break the notification invariant. Malfunction in this implementation, e.g., failure to turn on the indicator, will be detected and blocked by the invariant checks (the third component, explained below).

The second component is a trusted runtime monitor, which runs in the kernel or in the hypervisor. It intercepts the device drivers' attempts to write to the registers of sensors and indicators by removing the write permissions from the corresponding page table entries. It then consults with a set of de-

ployed invariant checks, which inspect the parameters of the write and decide whether, if allowed, the write would break the invariants corresponding to the sensor notifications or not. The monitor allows for the successful execution of the write only if it passes all the checks, and rejects it otherwise. In case of a reject, it blocks the write and force-reboots the system. §4 elaborates on the monitor.

The third component of our design is a set of invariant checks deployed in the monitor. To enable developing provably correct invariant checks, we present a high-level language for writing the sensor notification invariants. We also develop a verified compiler for the language using Coq that guarantees that the generated invariant checks maintain the semantic of sensor notification invariant. The compiler uses device specifications for inferring the device state transitions as a result of register writes. In addition, for off-chip devices whose registers are only accessible through a peripheral bus, we design a verified bus interpreter module that can infer the device register writes from the bus adapter register writes. §5 discusses the details of the invariant language, its compiler, and the bus interpreter module.

3.1 Threat Model

We have designed Viola to enforce sensor notifications' invariants in the system and prevent buggy or malicious code from breaking them. We will assume attackers with varying capabilities. The first attacker can only use the application API in the operating system, e.g., Android API. This attacker can run native code as well but without root privileges. The second attacker runs native code with root privileges in the user space, however cannot run code with kernel privileges or secretly modify the system image (for future boots). More specifically, in this case, we assume that the attacker cannot leverage the kernel vulnerabilities to inject code and assume that the kernel is configured to prevent a root user from easily modifying the running kernel memory. The latter is achieved by configuring the kernel to disallow loading of kernel modules and to avoid exposing the `/dev/kmem` file, which would allow user space code to have access to the kernel memory. Moreover, we assume that the mobile system implements the verified boot feature [3], which checks and verifies the integrity of the loaded system images. As a result, the attacker's attempt to modify the system images (i.e., the kernel and hypervisor images), which will take effect in future boots, will be detected and then blocked or at least communicated to the user with a notification at boot time. The third attacker leverages the vulnerabilities of the kernel to compromise it and hence can run code with kernel privileges. The fourth attacker is a more advanced version of the third attacker that, after compromising the kernel, leverages the vulnerabilities of the hypervisor to compromise it and hence can run code with hypervisor privileges. The fifth attacker is a root user in a system without the verified boot feature, which would allow him to rewrite the kernel and hypervisor images (to be used after a reboot). Finally, the sixth attacker has physical access to the device and can manipulate the hardware.

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

4. RUNTIME MONITOR

Viola's runtime monitor runs in the low-level system software, e.g., in the operating system kernel or the hypervisor, in order to monitor the device drivers' interactions with sensors and indicators. The monitor intercepts any attempts to write to device registers and feeds the write parameters to the invariant checks (§5). It allows the successful execution of a write only if it passes all the checks. Note that we focus on register writes, and not register reads, since the former is typically used to alter the state of a device. However, all the discussions are easily extended to register reads if they need to be monitored as well.

Using the kernel or hypervisor to insert the invariant checks provides a trade-off between trustworthiness, support for different operating systems, support for legacy systems, and performance. On the one hand, a hypervisor-based solution isolates the monitor from the operating system, which enhances its trustworthiness even against malware with complete control over the kernel execution (i.e., the third attacker in §3.1). It also makes the monitor implementation agnostic to the operating system, e.g., Android or iOS, or to the operating system version. The use of the hypervisor is further motivated by the fact that ARM processors have recently added virtualization hardware support [13] allowing an efficient implementation of the monitor in the hypervisor.

On the other hand, a kernel-based solution can be used in many existing mobile systems. This is because commercial mobile systems either currently do not have hardware support for virtualization or do not provide open access for programming the hypervisor. Also, inserting the checks in the kernel incurs less overhead to a register write compared to a hypervisor-based solutions (§7.2).

Note that an alternative layer for adding the invariant checks is the hardware. Such a solution is the most trustworthy as it minimizes the Trusted Computing Base to only the hardware. However, a hardware-based solution is costly as it requires additional circuitry per invariant. Moreover, such a solution is often practically impossible since the invariants span multiple I/O devices, e.g., camera and LED, which are typically integrated on and off the System-on-a-Chip (SoC) from various vendors in the form of closed source IP modules. Furthermore, while we want the notifications

to be unbreakable, it might be desirable to make them customizable by the user or the system designer, for example, if the user wants to intentionally turn off all notifications for a short period of time. Such customizations are not feasible with a hardware solution. In §8.1, we discuss how we can add support for secure customizations to Viola. Finally, note that, while we have not done so, if we employ our invariant checks in a verified kernel or hypervisor [20,34,36,38,44], we can reduce the TCB to only the hardware as well.

Viola's monitor intercepts device drivers' attempts to write to registers by removing the write permissions from the corresponding page table entries. This is feasible since in the ARM architecture all the registers are memory-mapped (i.e., Memory-Mapped I/O or MMIO). Depending on where the monitor is hosted, we use different page tables. First, for the hypervisor-based monitor, we leverage ARM's Stage-2 page tables [13,25]. In this case, a memory address is translated by two sets of page tables, one maintained by the operating system and one (i.e., Stage-2 page tables) maintained by the hypervisor. Removing the write permission from a Stage-2 page table entry forces writes to every register in the corresponding page to trap in the hypervisor, allowing the monitor to inspect them. Second, for the kernel-based monitor, we simply use the page tables used by the kernel. In this case, a write to protected register pages raises a page fault exception in the kernel.

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

Initially, we planned to identify and kill the responsible application upon detecting such page faults. However, doing so is challenging, as the violation might be triggered not by an application but by powerful malware with kernel or hypervisor privileges. Therefore, we believe that force-reboot is a safe approach to handle these attacks. Note that the force-reboot approach does not create an easily-exploitable and additional denial-of-service attack vector. This is because attackers with root, kernel, or hypervisor privileges can reboot the system using other methods anyway and the attacker without these privileges (i.e., the first attacker in §3.1) cannot trigger such page faults easily unless it circumvents the implementation of the sensor notification.

5. VERIFIED INVARIANT CHECKS

Viola's verified invariant checks receive the parameters of a register write as input from the monitor and decide whether the write, if executed, breaks any of the sensor notifications' invariants in the system. Unfortunately, manually developing checks that operate on the parameters of register writes is cumbersome and error-prone. An I/O device, i.e., a sensor or an indicator, might have several registers, each affecting the behavior of the device in some way. Moreover, a single register might contain a few device variables (i.e., the variable formed by a subset of bits in a register [45]), requiring bitwise operations in the invariant checks. Furthermore, the correct behavior of an I/O device might depend on other components as well, such as its power supply and clock source in the SoC, requiring Viola to monitor these components as well. Finally, Viola's monitor cannot directly monitor the writes to device registers for off-chip devices. It can only monitor the writes to registers of the peripheral bus adapters and must infer the device register writes.

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

Figure 3: The implementation of a microphone LED notification in Viola. For brevity, we have not shown the complete array of device variable states and the device specifications used in the code.

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

5.1 Viola's Invariant Language Syntax

Viola's invariant language leverages two main programming constructs. (*i*) *State* is used to specify the target states of the sensor or indicator, e.g., microphone recording or LED blinking. A device target state is a list of device variable states, where each device variable state is a 2-tuple containing a device variable and a value. As mentioned earlier, a device variable is the variable formed by a subset of the bits in a register [45]. As a hypothetical example, the three least significant bits of a register could form the device variable for camera resolution. These device variables must be defined in the device specifications (§5.2.1). (*ii*) *Binder* is finally used to bind the target states of the sensor and indicator.

Figure 3 shows an example invariant written in Viola for the microphone LED notification (LED blinking). In this example, we have first defined the target states of the sensor and indicator, `mic_rec` and `led_blink`, respectively. Each of these target states is defined as an array of device variable states, e.g., `led_blink_mode`. Moreover, in defining these target states, we have used the device specifications, i.e., `mic_spec` and `led_spec`, which provide the definition of each of the device variable states. Finally, the sensor notification invariant is defined by binding the target states of the sensor and indicator together.

5.2 Verified Compiler

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

Building and verifying a compiler is a cumbersome task. Our key idea to reduce this effort is to implement our compiler as a frontend for the formally verified CompCert C compiler [39], similar to the approach used by Wang et al. for Jitk [55]. The frontend compiles Viola's code to Cminor, an intermediate language in CompCert, which is then compiled to assembly by the CompCert backend. The assembly code is then translated to machine code using a commodity

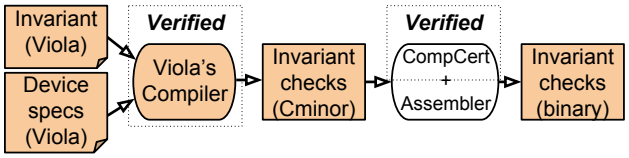

Figure 4: Compilation of Viola invariant code. Viola's compiler translates Viola code to Cminor code, which is then compiled to machine code using the verified CompCert compiler and an assembler. The darker components belong to Viola.

assembler since CompCert does not currently provide a verified assembler. Cminor is a simplified and typeless variant of a subset of C with support for integers, floats, pointers (but not the & operator), control constructs such as if/else (but not the goto statement), and functions (see [39] for more details).

Given that CompCert is a verified compiler, it preserves the semantics of the code in Cminor while generating the assembly code. Therefore, we only need to prove that our frontend maintains the semantics of Viola's code while generating the Cminor code. To achieve this, we implement the frontend in Coq and use Coq's interactive proof assistant [11] to prove the semantic preservation property. As will be explained, our compiler relies on device specifications to infer the device state transitions as a result of register writes. Figure 4 illustrates our approach.

We prove the semantic preservation property as follows. We formalize the I/O devices (i.e., sensors and indicators) as blocks of memory (MMIO) corresponding to their register spaces. We then prove, through forward simulation [41, 42], that both programs (i.e., the one written in Viola and its compiled Cminor program) return the same decision (i.e., allowing or rejecting a register write) given the state of these MMIO blocks, the invariant logic, and the device specifications. Put formally, we prove the following theorem.

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

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

Lemma *compile_step:* $\forall$ `S1 S2, Viola.step S1 S2` $\rightarrow$ $\forall$ `R1, match_states S1 R1` $\rightarrow$ $\exists$ `R2, plus Cminor.step R1 R2` $\wedge$ `match_states S2 R2.`

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

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

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

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

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

5.2.1 Device Specifications

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

Currently, we manually develop these specifications for the devices that we support. Fortunately, as observed by

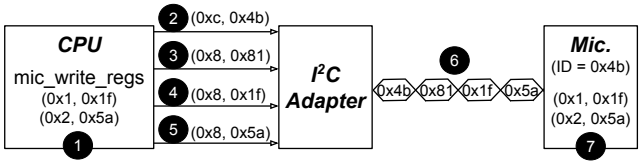

Figure 5: Simplified steps for writing to registers of a device over an I^{2}C bus. The numbers in parenthesis show register writes parameters (offset, value). The dark numbered circles represent the approximate progression of events.

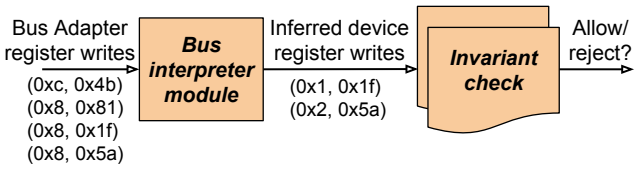

Figure 6: Bus interpreter module role. The bus register writes are intercepted in the monitor and passed to the bus interpreter module, which infers the register writes of the device and passes them to the sensor notification invariant checks.

Ryzhyk et al. [52], device specifications are increasingly available from hardware vendors as the same specifications are also used in the device design process. However, to leverage these specifications, a translator is needed to transform the existing specifications into the format supported by Viola.

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

5.3 Checks on Dependencies

Viola's goal is to guarantee that the indicator is on when the sensor is on. However, the indicator's state often has dependencies on other hardware components including the power supply and the clock source on the SoC that provide power and clock for the indicator. Therefore, the invariant logic to enforce in the system should be refined to `sensor on` → `indicator on` → `(indicator) power supply on` → `(indicator) clock source on`. Note that Viola does not monitor the sensor's dependencies since the failure to turn on the sensor does not violate the notification invariant.

We enforce this logical relationship using a chain of invariant checks all developed by Viola's invariant language and its compiler. In the example above, we develop three checks, one for `power supply on` → `clock source on`, one for `indicator on` → `power supply on`, and one for `sensor on` → `indicator on`. Moreover, in the monitor, we evaluate these checks in the aforementioned order to properly capture the dependencies.

5.4 Bus Interpreter Module

Registers of off-chip devices on a mobile SoC are not directly mapped into the CPU address space and are accessed through a peripheral bus, e.g., I^{2}C. Viola's monitor can only intercept and monitor the writes to the bus adapter registers. It must then "infer" the writes to the device registers.

Figure 5 illustrates a simplified, yet realistic, example. In order to write to two microphone registers (step 1), the CPU writes four values to the registers of the I^{2}C bus adapter, including the microphone device ID, a command byte, and two data bytes (steps 2 to 5). The I^{2}C adapter sends out these values on the bus to the connected devices (step 6). The microphone interprets the command byte as the register offset and the data bytes as the values to be written to consecutive registers (step 7).

Each device's interpretation of the same signal can be different, although most adhere to the same interpretation. For example, many devices support a continuous write mode (as in Figure 5): if the second most significant bit of the command value is set (i.e., the auto increment bit), the device then writes the following data bytes to consecutive registers starting at the register offset specified in the command.

Manually writing code to infer the device register writes can be challenging and error-prone. Hence, we present a *verified bus interpreter module* that achieves this goal. Figure 6 illustrates this module's role. The bus interpreter module receives the parameters of the register writes to the bus adapter and returns the inferred parameters of register writes to the device. The bus interpreter module is built from one or more state machines (in a daisy-chain). Different state machines can be created by providing "rules". Each rule operates on the input parameters and make modifications to the output parameters or to an accumulator. The output parameters are fed back to the machine allowing the rules to gradually (i.e., across multiple bus adapter register writes) infer the device register write. Similar to the invariant checks, we provide a high-level language for writing the state machine rules and implement a verified compiler for it. In the paper, we mainly use the word compiler to refer to invariant check's compiler, and not the bus interpreter module compiler, unless otherwise stated.

6. IMPLEMENTATION

The implementation of Viola consists of four components: the sensor notifications, the monitor, the invariant checks, and the bus interpreter module. Table 1 breaks down Viola's code base. Below, we elaborate on these components (except for sensor notifications, which are simple).

6.1 Runtime Monitor

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

Viola's monitor emulates the instructions as follows. It maps the register pages into a second set of virtual addresses,

Lang.	Total LoC	Component	LoC
Coq	6812	Invariant compiler	211
		Invariant language spec	251
		Invariant compiler proof	4812
		Device specs	282
		Bus interpr. compiler	126
		Bus interpr. language spec	203
		Bus interpr. compiler proof	879
		Bus specs (rules)	48
C	552	Monitor (Linux)	265
		Monitor (Xen)	223
		Cam. vib. notif.	10
		Mic. LED notif.	54

Table 1: Viola code breakdown.

which are used by the monitor only. Upon each page fault, it inspects the CPU registers to determine the value that was going to be written to the faulting address. It then issues an instruction to write this value to the same register using the second virtual address. Finally, it increases the program counter to point to the next instruction and returns.

It is important to note why the monitor must emulate the instruction rather than allowing it to execute natively. This is because the latter requires the monitor to – at least temporarily – enable the write permissions on the page table entry corresponding to the faulting address. However, doing this will create an attack vector given that modern mobile system hardware leverages multiple CPU cores, allowing the malware running on other cores to take advantage of this period of time to silently write to device registers. Emulation of the faulting instruction protects against such an attack as page table entries corresponding to the monitored register pages always remain read-only.

6.2 Verified components

We verify the functional correctness of the compilers for the invariant and bus interpreter languages. We implement and verify the compilers in Coq, which provides functional programming constructs in addition to logic constructs. The implementation of the invariant compiler translates the invariant logic to Cminor code and the implementation of the bus interpreter compiler translates the developer-provided rules to Cminor code. Figure 7 shows an example procedure from Viola's invariant compiler implementation. This procedure checks the input register offset against all the register offsets specified in the device specifications to find a match. It is implemented as a recursive procedure, which computes the inclusive disjunction (`Ebinop Oor`) of comparison (`Ebinop (Ocmp Ceq)`) of the input register offset (`reg_off`) with the list of registers in the specification (`regs`).

Once we develop and prove the functional correctness of the compilers, we generate their executable machine code. For this, we first use the code extraction facilities in Coq to generate equivalent OCaml code from the source code in Coq and then use the OCaml compiler to generate the machine code. Figure 8 illustrates these steps for our verified invariant language compiler. The same figure also shows that the implementation, the specification, and the proof are passed to the Coq proof checker for checking the correctness of the proof.

```
Fixpoint is_there_reg_match (regs : list register)
                            (reg_off : ident) : Cminor.expr :=
  match regs with
  | nil => Econst (Ointconst Int.zero)
  | hd :: tl => let _reg_off := hd.(reg_off) in
    Ebinop Oor (Ebinop (Ocmp Ceq)
      (Evar reg_off)
      (Econst (Ointconst (Int.repr _reg_off))))
        (is_there_reg_match tl)
  end.
```

Figure 7: A sample procedure from the implementation of Viola's invariant compiler.

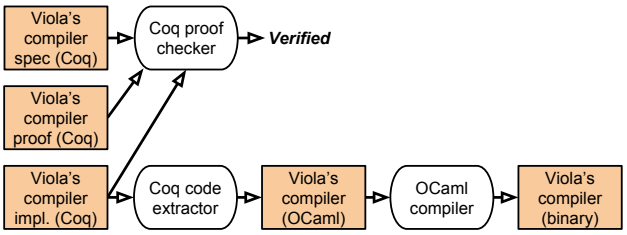

Figure 8: The compiler code is generated by first extracting the OCaml code from the Coq implementation and then compiling the OCaml code to machine code. The compiler's specification, implementation, and proof are also passed to Coq proof checker to verify the correctness of the proof. The darker components belong to Viola.

6.3 Supported Systems and Devices

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

We test Viola with the microphone and LED on Galaxy Nexus, with the camera and vibrator on Nexus 5, and (as a proof of concept) with GPIO LEDs on ODROID XU4. The audio chip (TWL6040) and the LED [10] in Galaxy Nexus are accessed through the OMAP4 I^2C bus, for which, we leverage our bus interpreter module. The rest of the devices are memory-mapped. Moreover, as discussed earlier, we currently only support continuous notifications (§2.2), which are not ideal for the vibrator. Yet, we use the vibrator in our implementation to demonstrate the applicability of Viola to a diverse set of indicators. We also implement the checks on the clock source dependency (§5.3) for the vi-

brator in Nexus 5. We do not, however, currently support the power supply for the vibrator due to its use of a message passing interface (see §8.2 for more details).

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

6.4 Trusted Computing Base

Here, we explain the trusted components in Viola.

Trusted components for the monitor. The trusted components for the monitor differ for the kernel-based and hypervisor-based implementations. The hardware and the monitor itself are trusted in both implementations. However, in the hypervisor-based monitor, the hypervisor is trusted, but not the kernel (including all its device drivers), which is trusted in the kernel-based implementation. Given that a hypervisor is typically smaller than an operating system kernel, our hypervisor-based implementation provides a smaller TCB. Moreover, while we currently use a commodity hypervisor (i.e., Xen), it is possible to use a smaller hypervisor to reduce the TCB. This is because we do not use the features of the hypervisor needed to run multiple virtual machines in the system, such as scheduling the virtual CPUs between virtual machines. Furthermore, as discussed in §4, it is possible to use a verified kernel or hypervisor [20, 34, 36, 38, 44] in the implementation, which will then reduce the TCB to only the hardware. Finally, note that in the kernel-based monitor, only the kernel is trusted, but not the user space including all of Android services and applications.

Trusted components for the verified compilers. In our compilers' implementation, the specifications that we develop for the languages' semantics, for the proof, for the devices, and for the buses (i.e., rules), are assumed to be correct. Moreover, the Coq's simple proof checker that verifies the correctness of the proof is trusted but not the tactics used in the construction of the proof. Also, the Coq extraction facilities and the OCaml compiler are trusted. In addition, we leverage an existing assembler in the backend of our compiler and hence the assembler needs to be trusted. However, if a verified assembler is added to CompCert [39], we can simply leverage it in our solution without requiring any further engineering effort. We can also leverage other solutions for verifying assembly programs [28, 29, 54, 58].

While bugs in any of these trusted components can undermine the functional correctness of the compilers, we believe that Viola provides more reliable guarantees compared to the alternative of (*i*) not using invariant checks or (*ii*) developing the checks manually. In (*i*), the whole operating system and the applications will be trusted. In (*ii*), not only the implementation of the checks can have bugs and need to be trusted, the compiler for the language used, e.g., gcc, must be trusted as well.

7. EVALUATION

Next, we evaluate the engineering effort needed to use Viola and its performance and power consumption overheads.

7.1 Engineering Effort

For one to use Viola, one must develop the sensor notification, Viola code corresponding to the notification invariant (e.g., Figure 3), the specifications for the devices and, if needed, the rules for the peripheral buses. The first two are quite straightforward. However, the last two can be cumbersome depending on the devices and the buses. However, as mentioned in §5.2.1, often partial specifications are enough in Viola, which significantly reduces the required effort for device specifications. Moreover, device specifications are increasingly available from hardware vendors [52]. We plan to support these specifications in the future with a translator, which then reduces the effort needed by the developer. Finally, writing the specification for a device or the rules for a bus is a one-time engineering effort. The same device specification and especially the same bus rules can then be reused for various sensor notifications.

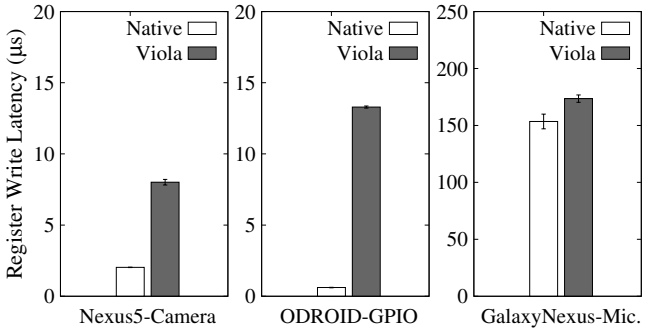

Figure 9: Register write latency. Note that the rightmost figure uses a different y-axis range. Also, the figure in the middle is based on a hypervisor-based monitor, whereas the other two figures are based on a kernel-based monitor.

7.2 Performance

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

Figure 9 summarizes the results. It shows the average and standard deviation of monitored register write latency in these systems. It demonstrates that the added latency is significant. Part of this latency is due to the page fault exception and part is due to the code running in the fault handler including a large part of the runtime monitor's code (such as the code that inspects the CPU registers) and the invariant checks.

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

Another observation is that the register write latency for microphone on Galaxy Nexus is much higher than the rest. This is because the microphone registers are accessed through the I^2C peripheral bus, which is not only slower than a memory-mapped access, it also requires multiple writes to the bus adapter registers.

Macrobenchmarks. We measure the overhead of Viola on the performance of sensors.

First, we measure the performance of the camera on the Nexus 5 in terms of the framerate. We measure the framerate for varying resolutions with and without Viola. For each

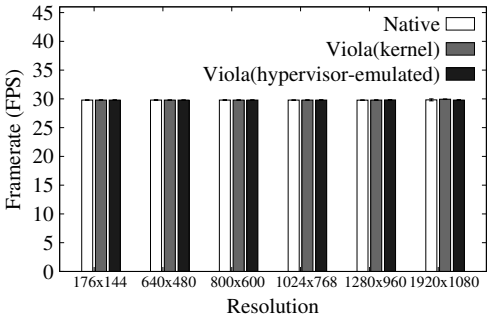

Figure 10: Camera's performance in Nexus 5 for both kernel-based and hypervisor-based monitors. Note that we emulate the overhead of the hypervisor-based monitor in Nexus 5 (§7.2).

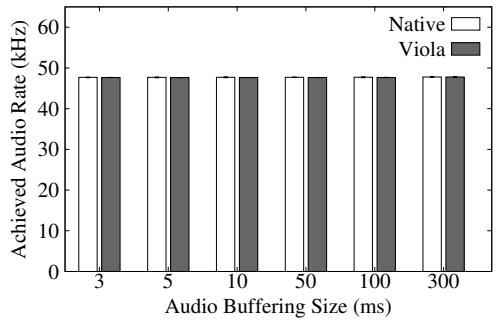

Figure 11: Microphone's performance in Galaxy Nexus based on a kernel-based monitor.

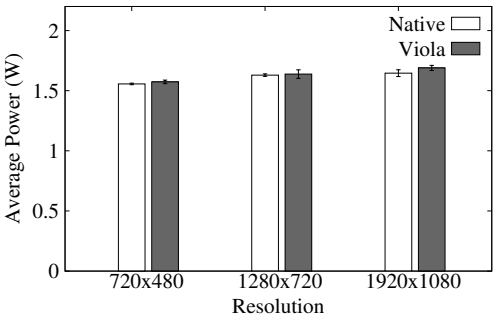

Figure 12: Power consumption of Nexus 5 when running a video recording application [1] in the background and with the display off. The results are based on a kernel-based monitor.

resolution, we measure the average framerate achieved over a 1000 frames. We discard the first 50 frames to avoid the camera initialization overhead and repeat each experiment three times. Figure 10 shows the results. It demonstrates that Viola's overhead on the camera performance is negligible irrespective of the resolution.

The reason for this negligible performance overhead is the infrequency of register writes. To demonstrate this, we measure the number of register writes that are intercepted by Viola when the camera is on. We find that there is one register write interception every 19.3 ms, which is significantly larger than the latency of a single register write (i.e., 2 and 8 μs for native and Viola). The number of register writes are relatively small since they are used mainly for sending control commands to the camera and not for exchanging data. Moreover, in the case of camera, Viola does not intercept writes to all the registers (see discussion on partial specifications (§5.2.1)). Only writes to the registers in the same register page as those monitored by Viola are intercepted. Our experiment shows that this constitute 4% of all register writes.

Moreover, we demonstrate that even the more significant register write latency incurred by a hypervisor-based monitor will not impact the performance of the camera. For this, we emulate the overhead of the hypervisor-based monitor in Nexus 5 by artificially adding a delay of about 6.7 μs to our fault handler since this is the additional latency incurred by the hypervisor as derived from the ODROID XU4 results in Figure 9. We then measure the camera framerate and show the results in Figure 10, which demonstrate almost no performance overhead. This is, similarly, due to the infrequency of register write interceptions.

Second, we measure the performance of the microphone on Galaxy Nexus in terms of the audio rate. We measure the audio rate for a one-minute recording experiment and with varying audio buffering sizes, which determines the audio latency. We repeat each experiment three times. Figure 11 shows the results. Similar to camera, we notice that Viola's overhead on the microphone performance is negligible. We also measure the number of intercepted register writes for the microphone and find them to be very small, i.e., only about 20 when starting the capture.

7.3 Power Consumption

Battery lifetime on mobile systems is one of the most important usability metrics. Therefore, it is important that Viola does not enhance user's privacy at the cost of reduced battery lifetime. We therefore measure the power consumption of the system with and without monitoring by Viola. We measure the power consumption using the Monsoon Power Monitor [9].

First, we measure the power consumption of Nexus 5 when recording a video. To magnify the relative overhead by Viola, we put the camera application [1] in the background and turn the display off. Figure 12 shows the results. It demonstrates that Viola incurs additional power consumption but that the overhead is small (less than 45 mW). Note that the high baseline power consumption of the smartphone is mainly due to the CPU and camera being on.

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

8. LIMITATIONS AND FUTURE WORK

8.1 Sensor Notification Customizations

It is critical that the sensor notifications cannot be deactivated by attackers. This has motivated us to design a solution that enforces the notifications unconditionally. However, we envision scenarios that the user might need to disable the notifications temporarily, e.g., if light, vibration, or sound interferes with user's activities. Allowing customizations while protecting against malware raises new challenges that we plan to address in future work. One key idea is to provide an orthogonal channel (to the operating system) in order to deploy and update the invariant checks. One candidate channel is a minimal virtual machine, which can be interacted with only if the user presses a certain key combination, enforced by a few permanent invariants.

8.2 Message-Passing Hardware Interfaces

The control commands to most I/O devices and hardware components are programmed using a set of registers. However, some devices and components adopt a message-passing interface for control commands, where they exchange messages with the CPU using shared memory buffers. We have encountered two such devices: the image subsystem in Galaxy Nexus (which includes the camera) and the voltage regulators in Nexus 5 (one of the vibrator' dependencies (§5.3)). Viola does not currently support these devices. To overcome this limitation, we plan to extend our bus interpreter module to support message-passing interfaces.

8.3 Speaker and Display as Indicators

In this paper, we demonstrated the use of LED and vibrator as indicators. Other possible indicators are the speaker and display. However, the use of these indicators in Viola present important challenges that we plan to address in our future work. First, for both indicators, the *data passed to the device* must be inspected and verified, e.g., audio samples and display pixels. Doing so requires support in the monitor to intercept and inspect the Direct Memory Access (DMA) operations. It also requires support in Viola's language to allow the developer to concisely specify the intended data. Second, in the case of the speaker, the time that each audio sample is played must also be monitored and verified to be correct. This is because tampering with the timing of the samples can affect the effective sounds heard by the user.

8.4 Context-Sensitive Notifications

The best indicator to use might depend on the mobile system's context. For example, if the ambient noise level is high, vibration may be more effective than a beep. Viola can be extended to support context-sensitive notifications as well. For this, Viola needs to monitor various sensor readings and enforce the indicators accordingly.

8.5 Two-Way Notifications

Two-way notifications require the sensor and indicator to be in their target states only simultaneously (§2.2). This is challenging as the states of the sensor and indicator have to change atomically. We plan to solve this challenge by implementing *verified I/O transactions*, in which Viola's monitor buffers several register write parameters, and execute all of them atomically.

8.6 Temporary Notifications

While some types of indicators, such as LED, should stay on as long as the sensor is on, other indicators are best if temporary, e.g., a short vibration or a brief beeping sound (§2.2). Implementing temporary notifications requires the invariant checks to store timing information. Moreover, in case the indicator is not turned off after the specified period of time, the monitor must turn it off on its own. Implementing such behavior in the monitor requires code synthesis from the device specifications and is part of our future work.

9. RELATED WORK

9.1 Untrusted Operating Systems and Drivers

Applications rely heavily on the correctness of the operating systems. However, contemporary operating systems are large, complex, and full of bugs, allowing malicious applications to compromise them. This observation has fueled research into protecting the applications from a compromised operating system. Most solutions, such as Overshadow [22] and InkTag [35], do so with a trusted hypervisor, which mediates applications' interactions with the operating system. However, due to the wide and complex interface between an application and the underlying operating system, these solutions cannot protect the applications against all possible attacks, such as Iago attacks [19]. Similar to this line of work, we leverage the hypervisor in Viola in order to make the operating system untrusted. However, while the focus of this line of work is on protecting the application from the operating system, Viola's focus is on the correctness of sensor notifications.

Nexus [56] makes the device drivers untrusted by running them in user space and by vetting their interactions with the I/O devices. Similar to Viola, Nexus adopts a domain-specific language to write specifications for the devices, which are then compiled into reference monitors used in the kernel. Unlike Nexus, Viola's compiler is formally verified. Moreover, Viola's invariant checks monitor more than one I/O device (i.e., a sensor and an indicator) and enforce a relationship between their states. Unlike Nexus, which implements the monitor in the kernel, we have demonstrated an implementation of our monitor in the hypervisor as well. Finally, while Nexus is implemented for the x86 architecture and PCI devices, Viola is implemented for the ARM architecture and devices that are directly accessed by the CPU or through an I^2C bus. Despite these differences, Viola can benefit from the design of Nexus by moving the device drivers to user space. Such a design will enhance the security guarantees of Viola's kernel-based monitor as it removes the device drivers out of the TCB.

9.2 Virtual Machine Introspection

Virtual machine introspection (VMI) uses the hypervisor to monitor the operating system for intrusion detection [26, 30, 32]. It provides good visibility of the operating system internals while protecting the monitoring system from the attacks on the operating system. Similar to VMI, we leverage the hypervisor to enforce the I/O invariants eliminating the need to trust the operating system. Our work is, however, fundamentally different as, unlike VMI, we provide guarantees on the correct behavior of I/O devices.

9.3 Software Verification

An alternative approach to Viola for implementing trustworthy sensor notifications is to guarantee that there are no bugs in the whole mobile operating system. A large amount of work has tried to face such a challenge head-on by finding and eliminating bugs in existing software using static analysis [15] and model checking [24, 47]. These solutions have an important limitation: they do not scale to large software systems, e.g., the whole Android code base. Moreover, some of these solutions might not be practical as important parts of the I/O stack in mobile operating systems, including Android, are closed source.

Our key insight in Viola is that we can *enforce sensor notification invariants in low level system software* eliminating the need to verify the correctness of the whole operating system. However, it is important to note that existing solutions discussed above are more generic than ours. They can find various types of bugs in software systems or can be used to find violations of arbitrary invariants. Our solution is specific to sensor notifications.

More specifically, several existing solutions improve the quality of device drivers by finding or avoiding driver bugs either through static analysis [14,23,48], better interface language [45], synthesis [51, 52], or symbolic execution [37, 50]. We note that this line of work reduces the probability of the violation of sensor notification invariants, but on their own, these solutions are not adequate to guarantee the invariants for two reasons. First, the device driver is only part of the I/O stack, and the invariant can be violated as a result of bugs in other components of the stack, e.g., the I/O system services in Android. Second, existing solutions are often best efforts to eliminate bugs but cannot eliminate all the bugs nor provide formal guarantees about an invariant.

Finally, our verified compiler for Viola (§5.2) is related to existing work on verified compilers [39], verified kernels and hypervisors [20, 34, 36, 38, 44], and verified file systems [21]. Indeed, our compiler is built on top of a verified compiler (i.e., CompCert [39]). However, unlike Viola, none of these solutions provide formal guarantees about the behavior of I/O devices.

9.4 Information Flow Control

Systems such as TaintDroid [27] and Panorama [57] can track the flow of information from sources that produce sensitive information, e.g., camera and microphone, to sinks that can leak the information, e.g., network interface card. Such systems can therefore be used to notify the user when sensor data propagate to sensitive sinks. However, there are four important differences between such notifications and Viola. First, Viola is designed to reliably detect when sensors are turned on and off. To do this, Viola uses device specifications of sensors and monitors writes to registers of these devices. In contrast, information flow control systems cannot reliably detect the states of the sensors. Instead, given some sensor data stored in memory, even if the data were captured legitimately with the user's approval, information flow control systems can reliably track the propagation of data and potentially inform the user if the data leave the mobile system. In this sense, Viola and information flow control systems can provide complementary forms of notifications, i.e., notification about when the sensor is on vs. notification about when the sensor data are about to leave the mobile system. Second, information flow tracking systems incur noticeable performance overhead in the system as they need to instrument and track several processor or high-level language instructions. On the other hand, Viola's overhead is negligible since it only monitors infrequent writes to registers of I/O devices. Third, Viola supports a hypervisor-based implementation, which will make the operating system fully untrusted. While some information flow control systems are implemented in the hypervisor as well, e.g., Panorama [57], others are implemented at higher layers of the stack, e.g., Java Virtual Machine (JVM) in case of TaintDroid [27], resulting in a larger TCB. Finally, unlike some information flow control systems that have dependency on the JVM implementation (e.g., TaintDroid), Viola has dependency on the kernel or hypervisor.

9.5 Other Related Work

Operating systems have employed kernel interpreters for syscall monitoring, e.g., Linux Seccomp, or packet filtering [43]. Jitk [55] provides a trustworthy kernel interpreter using Coq. Our solution can also be considered as filtering the states and behavior of sensors and indicators in mobile systems. Indeed, our approach in how we use Coq to verify the functional correctness of our compilers has been influenced by Jitk. However, unlike Jitk that performs syscall and network filtering and socket monitoring, Viola monitors the operating system's interactions with I/O devices.

Bianchi et al. implement a notification on the Android system navigation bar to notify the user of the origin of the application she is interacting with [16]. Our work on Viola is in line with this work in terms of motivation. However, unlike our solution, their solution does not provide any guarantees on the correctness of the notification.

Trusted sensors provide mechanisms for maintaining the integrity of sensor readings in mobile systems for the user of the data [33, 40, 53]. This line of work is orthogonal to Viola, which is concerned with immediate and unconditional feedback to the user about the usage of sensors, and not with the integrity of the sensor data after capture.

10. CONCLUSIONS

We presented Viola, a system aimed at enhancing mobile systems users' privacy by providing trustworthy sensor notifications, which unconditionally inform the user when the sensors are on. Viola uses verified invariant checks to inspect the writes to sensors' and indicators' registers in the kernel or in the hypervisor, and rejects those that violate the sensor notifications' invariants. We reported an implementation of Viola on various mobile systems, including LG Nexus 5, Samsung Galaxy Nexus, and ODROID XU4, and for various sensors and indicators, such as camera, microphone, LED, and vibrator. We demonstrated that Viola's overheads to the sensor's performance and system's power consumption are negligible and small, respectively. We believe that trustworthy sensor notifications are critical for mobile systems, which are increasingly capable of recording private information about their users.

Acknowledgments

The authors thank their shepherd, Dr. Eduardo Cuervo, and the anonymous reviewers for their insightful comments. The authors also thank Justin A. Chen for his help in revising the paper.

11. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. ACM EuroSys*, 2006.

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

[16] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[32] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2003.

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

[34] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S. Weng, H. Zhang, and Y. Guo. Deep Specifications

and Certified Abstraction Layers. In *Proc. ACM POPL*, 2015.

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

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

[37] V. Kuznetsov, V. Chipounov, and G. Candea. Testing Closed-Source Binary Device Drivers with DDT. In *Proc. USENIX Annual Technical Conference*, 2010.

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

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

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

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

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

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

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

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

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

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

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

[49] R. S. Portnoff, L. N. Lee, S. Egelman, P. Mishra, D. Leung, and D. Wagner. Somebody's Watching Me? Assessing the Effectiveness of Webcam Indicator Lights. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, 2015.

[50] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing Drivers without Devices. In *Proc. USENIX OSDI*, 2012.

[51] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic Device Driver Synthesis with Termite. In *Proc. ACM SOSP*, 2009.

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

[53] S. Saroiu and A. Wolman. I Am a Sensor, and I Approve This Message. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2010.

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

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

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

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

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