Bai *et al. EURASIP Journal on Information Security* (2017) 2017:10
DOI 10.1186/s13635-017-0061-8

EURASIP Journal on Information Security

RESEARCH Open Access

Sensor Guardian: prevent privacy inference on Android sensors

Xiaolong Bai*, Jie Yin and Yu-Ping Wang

Abstract

Privacy inference attacks based on sensor data is an emerging and severe threat on smart devices, in which malicious applications leverage data from innocuous sensors to infer sensitive information of user, e.g., utilizing accelerometers to infer user's keystroke. In this paper, we present Sensor Guardian, a privacy protection system that mitigates this threat on Android by hooking and controlling applications' access to sensors. Sensor Guardian inserts hooks into applications by statically instrumenting their APK (short for Android Package Kit) files and enforces control policies in these hooks at runtime. Our evaluation shows that Sensor Guardian can effectively and efficiently mitigate the privacy inference threat on Android sensors, with negligible overhead during both static instrumentation and runtime control.

Keywords: Android, Privacy, Protection, Sensor, Instrumentation

1 Introduction

With the proliferation of smartphones, users are requiring various functionalities for different needs. Smartphones now are not just used for calling and messaging, they are utilized in multiple ways, such as entertainment, social networking, finance, and traveling. With these multi-purpose requirements, smartphones are designed with additional features, in both hardware and software. In particular, current smartphones are equipped with various sensors, like accelerometer and gyroscope, in the need of sensing device movement and ambient environment to make apps (short for applications) involved in user activity. For example, a navigation app can use the orientation sensor to suggest user about driving directions, and accelerometer can be used by a game app for user to steer a car by rotating the device.

Some sensors, like accelerometer and gyroscope, seem less sensitive compared to other device features like GPS location, device ID, and network access. Mainstream mobile platforms, like Android, set no restrictions on accessing these innocuous sensors. However, when a user uses the device in regular patterns, things become more complicated. For example, when the user touches the screen to enter her screen lock password, the accelerometer may reveal regular patterns [1]. Recent studies [1–10] have shown that, by utilizing these innocuous sensors, attackers are able to infer users' behavior patterns and launch severe privacy inference attacks, such as keylogging [1, 2], inferring user activity [4], profiling [6], and tracking [9]. For example, [4] demonstrates the feasibility for a malicious app to infer the smartphone user's transportation mode, i.e., whether the user is stationary, walking, running, biking, or in motorized transport, by utilizing the onboard accelerometer. Indeed, privacy inference attacks based on innocuous sensors is an emerging and severe threat on smart devices.

More specifically, the problem of "privacy inference based on sensors" (PIS) here refers to any malicious activities that involve collecting data from those sensors that are not protected by mobile systems, in an attempt to infer sensitive user information. We define those innocuous sensors that are not protected by the Android permission system as unrestricted sensors or unprivileged sensors. In this paper, we present a privacy protection system, named Sensor Guardian, to address the PIS problem on Android by hooking apps' sensor-related API (short for application programming interface) calls and preventing apps from arbitrarily accessing those unprivileged sensors. In particular, Sensor Guardian uses static instrumentation techniques to insert hooks into apps' code and hook both Java and native API calls since apps can access sensors in both ways. At runtime, those inserted hooks control apps' access to sensors depending on various control

*Correspondence: bxl12@mails.tsinghua.edu.cn
Department of Computer Science and Technology, Tsinghua University, Beijing, China

policies. We conducted several experiments to evaluate Sensor Guardian's effectiveness and performance. The result shows that Sensor Guardian can effectively solve the PIS problem with negligible overhead, which is quite improved compared to prior protection techniques for the PIS problem. The contributions of our paper are outlined as follows:

1. We propose a new privacy protection system, Sensor Guardian, to address the PIS problem. Sensor Guardian uses static instrumentation techniques to hook and control apps' API calls of accessing unprivileged sensors. To the best of our knowledge, Sensor Guardian is the first to use static instrumentation techniques to solve the PIS problem.
2. We evaluated Sensor Guardian's performance. It incurs negligible overhead during both static instrumentation and runtime control. Compared to prior protection techniques, its performance is quite improved, especially in CPU, memory, and power consumption. With the help of Sensor Guardian, we also reviewed how and why sensors are used in apps.

Roadmap The rest of the paper is organized as follows: Section 2 introduces the background information related to our research; Section 3 presents our privacy protection system Sensor Guardian; Section 4 reports the evaluation of Sensor Guardian; Section 5 discusses Sensor Guardian's strengths, weaknesses, and future work; Section 6 reviews related prior works; and Section 7 concludes the paper.

2 Background

In this section, we first introduce how an Android app is built up. Then, we introduce the sensors supported in different mobile operating systems and how apps access sensors on Android, including some critical APIs that will be hooked in Sensor Guardian. We will also introduce basic procedures in static instrumentation techniques that are adopted in Sensor Guardian. At last, we introduce the adversary model that Sensor Guardian defends against.

2.1 Android app

Android apps are mostly written in Java. To build an Android app, its source files are first compiled into Java bytecode, which is then translated to a specific bytecode format called Dalvik bytecode. All bytecode files are then stored in a dex (short for Dalvik Executable) file. Besides Java code, developers can also employ native code, like C and C++, to Android apps, in order to perform self-contained and CPU-intensive operations. Those C and C++ source files are eventually compiled into shared libraries. In addition to the dex file and native shared libraries, an Android app also requires a manifest file, named AndroidManifest.xml, to describe its name, components, required permissions, and other important information. The manifest file, dex file, shared libraries, and some other resources are further packaged into a single zip-compatible archive file called Android Package Kit (APK). After it is signed by the developer's certificate, an APK file can finally be deployed to Android devices to install the Android app.

2.2 Sensors

Smart devices provide several sensors to enable apps to monitor the motion, orientation, and various environmental conditions of devices. All mobile platforms, including Android, iOS, and Windows Phone, provide APIs for developers to access data from these built-in sensors. We studied the types of their supported unprivileged sensors by reading these platforms' official documents for developers [11–13]. Table 1 shows the unprivileged sensors that are available for developers in the latest versions of these platforms (Android 7.1, iOS 10.0.2, Windows Phone 10). Specifically, Android supports 25 sensors, iOS supports 6, and Windows Phone supports 13. Given that Android supports more kinds of unprivileged sensors than other platforms and most of prior attacks [1, 2, 4, 7, 10] are implemented on Android, in this paper, we focus on the PIS problem on Android platform. Though the availability of some sensors may vary between different devices and versions, most Android devices have some basic built-in sensors like accelerometer and magnetometer. These basic sensors are still enough to infer a lot of sensitive information [1–4, 7, 8, 10].

Android provides a sensor framework [14] to help developers access and manage data from supported unprivileged sensors. To access a sensor with the Java APIs in this framework, an app needs to first get an instance of `SensorManager` and check whether the sensor is available on the device. If available, the app can get corresponding `Sensor` instance and register a `SensorEventListener` for emerging

Table 1 Sensors supported by different mobile platforms

Android	Accelerometer, magnetic field, orientation, pressure, temperature, linear accelerometer, rotation vector, relative humidity, ambient temperature, game rotation vector, significant motion, step detector, step counter, magnetic field uncalibrated, proximity, geomagnetic rotation vector, gyroscope, gravity, heart rate, heart beat, gyroscope uncalibrated, motion detect, light, pose 6DOF, stationary detect
iOS	Accelerometer, gyroscope, magnetometer, altimeter, attitude, pedometer
Windows Phone	Accelerometer, gyroscope, magnetometer, altimeter, barometer, inclinometer, orientation, pedometer, proximity, simple orientation, activity, light, compass

sensor events by calling the listener registration function `registerListener()` on this sensor instance. By implementing the callback of the listener, `onSensorChanged()`, the app can get incoming sensor events and retrieve sensor data from the events in the callback.

In addition to these Java APIs, this framework also provides native APIs for apps to access sensors using C or C++. Similar to the corresponding Java APIs, the app using native APIs also needs to get the instance of `SensorManager`, check availability of the desired sensor, and register for events from the sensor. Instead of implementing listener callbacks, when using native APIs, the app needs to register for sensor events with a sensor event queue and poll this queue by calling the function `ASensorEventQueue_getEvents()` to retrieve new events and read data.

On Android, sensors are managed by the SensorService system service; apps in user mode need to communicate with this service through inter-process communication (IPC) to access sensors. The main IPC method on Android is Binder [15]. In fact, the aforementioned Java and native APIs provided by the sensor framework abstract the IPC details and manage several internal data structures for developers. Specifically, during registration, the framework keeps a mapping between the sensor type and the registered listener or event queue, communicates with SensorService through Binder to enable the sensor, and waits for sensor events sent by SensorService. Apps do not need to care about these details; they only need to call the registration functions and concentrate on how to handle the received sensor events.

2.3 Static instrumentation

Sensor Guardian employs static instrumentation techniques to hook and control apps' sensor-related API calls. In general, these techniques modify apps' APK files and add new code into these files before their installation. The new code replaces the original API calls with calls to new functions, and these functions decide whether the app is allowed to perform the original calls. At runtime, the modified APK file is installed on the Android device instead of the original one. Since Android apps can have both Java and native code, static instrumentation techniques can be categorized into Java Instrumentation [16, 17] and Native Instrumentation [18]. Sensor Guardian combines these two types of instrumentation techniques, given that Android provides both Java and native APIs to access sensors. During implementation, we further addressed several technical issues and made some improvements to the existing static instrumentation techniques, to achieve complete protection with low overhead.

2.4 Adversary model

We assume that the attackers are the third-party malicious apps installed on users' Android devices. These apps attempt to stealthily collect data from unprivileged sensors on the device without any permission, in order to infer sensitive user information. We only focus on those sensors that are not protected by permissions, because malicious apps leveraging these sensors cannot be detected or defended by the existing protection mechanisms on Android. We assume that the malicious apps only run in the user mode without any system privileges, as most real-world Android malwares do. With the confinement of sandbox, the unprivileged malicious app cannot directly read or write sensor device files to retrieve sensor data. It can only collect sensor data through APIs provided by the system. Given that the main threat of the PIS problem is from these user-mode malicious apps, we trust the Linux kernel, Android platform, Android's application sandbox mechanism.

3 Sensor Guardian

In this section, we first introduce Sensor Guardian's architecture. And then, we introduce the implementation details of Sensor Guardian's static instrumentation and runtime control. The control policies that Sensor Guardian currently supports are also introduced in this section, as well as how Sensor Guardian can be deployed.

3.1 Architecture

Sensor Guardian consists of two parts: Instrumentor and Policy Manager. Instrumentor statically instruments an app's APK file and inserts hooks in its Java and native code. Policy Manager is a standalone Android app in user mode without any system privilege. Currently, we implement Instrumentor on a regular personal computer (PC). When an app is instrumented and installed on an Android device, it needs to communicate with Policy Manager on the same device to get up-to-date control policies. Policy Manager manages control policies for all instrumented apps on the same device and helps them make control decisions.

3.2 Implementation

To ensure its effectiveness, efficiency, and complete protection, Sensor Guardian overcomes several challenges and makes some improvements to the existing techniques both in static instrumentation and runtime control.

As illustrated in Fig. 1, we elaborate the implementation details of Instrumentor and Policy Manager separately in this section. For Instrumentor, the implementation details of both Java and Native Instrumentation are explained. In Java Instrumentation, we first explain how sensor-related Java APIs are hooked and then explain how we handle Java reflection, a mechanism for apps to dynamically

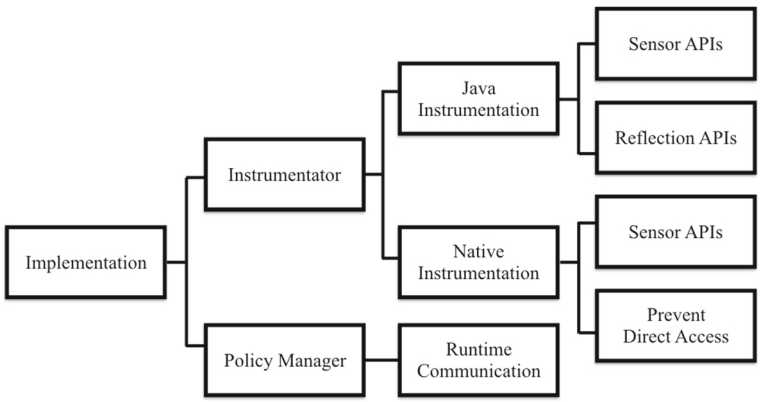

Fig. 1 Overview of Sensor Guardian's implementation. The overview of Sensor Guardian's implementation, including its Instrumentor and Policy Manager

change their runtime behaviors. In Native Instrumentation, besides the sensor APIs that are hooked, we also elaborate how we prevent apps from directly accessing sensors, i.e., not using the system-provided APIs. For Policy Manager, the main implementation detail that needs to be explained is how Policy Manager communicates with instrumented apps to enforce control policies.

Java Instrumentation Java Instrumentation (or bytecode rewriting) techniques generally undergo three steps as disassembling, modifying assembly code, and reassembling [16, 17, 19]. They make modifications to the intermediate assembly code files (like smali [20]) generated by disassembling tools (like apktool [21]) and reassemble the modified code into a new dex file. In Sensor Guardian, we do not rely on these disassembling tools. Instead, we modify and use the Dalvik bytecode library dexlib2 [22] to directly instrument Android apps at Dalvik bytecode level. The modification to dexlib2 mainly addresses the problem that some internal data structures in dexlib2 cannot be directly converted to raw Dalvik bytecode. Compared to prior static instrumentation techniques, our Instrumentor does not need to generate intermediate assembly code files, which saves both time and space during instrumentation.

We take three steps to hook a specific API in an app as follows:

1. Define a wrapper function that takes the same parameters as the specific API.
2. Look for `invoke` instructions (in the app's dex file) whose invocation target is the specific API.
3. Replace the invocation target with the wrapper function.

In terms of hooking apps' access to sensors, we hook those critical sensor-related APIs mentioned in Section 2.2. In particular, we first hook the listener registration function, `regsiterListener()` in `SensorManager`. In the wrapper function (say $\text{wrapper}_{\text{reg}}$) of this registration function, we register a proxy sensor event listener instead of the original one. In the proxy listener, we implement the corresponding callback function, `onSensorChanged()`, to intercept incoming sensor events. When an event is intercepted, this proxy's callback decides whether and how to forward the intercepted sensor events to the original listener's callback depending on current control policies.

Reflection Reflection is a Java mechanism for apps to examine or modify their runtime behaviors. Apps can use reflection to dynamically change their call targets, which could help them bypass static analysis. To prevent apps from retrieving sensor data in this implicit way, our Instrumentor employs the same strategy in [16], that is, we hook the critical reflection function `invoke()`[1] in class `java.lang.reflect.Method` (in short, `Method`) [23, 24]. As stated above, when we hook a specific API, we replace the API call with a call to its corresponding wrapper function. We name the `invoke()` API's wrapper as $\text{wrapper}_{\text{invoke}}$. In $\text{wrapper}_{\text{invoke}}$, we check whether the function being invoked by reflection is the critical sensor-related API, i.e., the listener registration function `regsiterListener()` in `SensorManager`. In specific, when $\text{wrapper}_{\text{invoke}}$ is called, besides the `invoke()` API's original parameters, a `Method` object (say $\text{object}_{\text{method}}$) is also passed to $\text{wrapper}_{\text{invoke}}$, which represents the method being invoked. From $\text{object}_{\text{method}}$, we can retrieve the class name and method name of the method being invoked (with APIs like `getDeclaringClass()` and `getName()` [24]) and then check whether the class name is `SensorManager` and the method name is

`regsiterListener()`. If they are, then we invoke $\text{wrapper}_{\text{reg}}$ instead of the original registration function.

Native Instrumentation Besides Java APIs, Android also provides native APIs for apps to access sensor data (see Section 2.2). Instrumentor employs the method in Aurasium [18] to hook native API calls, which examines the native shared libraries loaded by the app and replaces function pointers in their global offset table (GOT). Aurasium examines shared libraries only at the start of an app, but many third-party libraries using sensors, like libunity [25], may be loaded dynamically at any time during the app's runtime. In Sensor Guardian, besides at an app's start, we examine a library at any time the library is dynamically loaded. Android apps can load a native library by calling either `System.loadLibrary()` in Java or `dlopen()` in native code. We hook these loading functions. Once an app is loading a library at runtime, we know the library name and loaded address, and then, we examine the GOT of the loaded library and replace the function pointers of sensor-related APIs (in GOT) with their corresponding wrapper functions' addresses.

To retrieve sensor data using native APIs, an app needs to register for sensor events with a sensor event queue and call the function `ASensorEventQueue_getEvents()` to poll this queue for new sensor events (see Section 2.2). We hook these registration and polling functions by replacing their function pointers (in the GOTs of the app's loaded shared libraries) with their wrapper functions' addresses. The registration function's wrapper records the registered sensor type and the queue to receive sensor events, while the polling function's wrapper intercepts incoming sensor events and depends on current control policies to decide whether it should forward the intercepted sensor events to the app's registered sensor event queue.

An app can also use `dlopen()` to open `libandroid` in which the sensor-related APIs are defined and use `dlsym()` to get the function pointer. We hook both `dlopen()` and `dlsym()`, record the address of `libandroid` returned by `dlopen()`, and check whether the app is using `dlsym()` to resolve the address of the sensor-related APIs (the abovementioned registration function and queue-polling function). Instead of returning the original APIs' addresses, we return the addresses of these APIs' wrappers.

Prevent direct access to sensors As mentioned in Section 2.2, the sensor-related APIs eventually use Binder IPC to communicate with the SensorService system service and acquire sensor data. Hao et al. [19] showed the possibility that Android apps can access system services directly through IPC to evade API-level access control. To prevent such an evasion, we hook the Binder IPC to prevent apps from accessing sensors in such a direct way. We hook the native function `ioctl()` in libc.so to intercept the Binder IPC. On Android, all IPCs are sent through this function. We reconstruct the high-level IPC communication in `ioctl()` using the same method in Aurasium. When we observe an IPC of enabling a sensor, we examine the current Java and native call stacks of the app to check whether this IPC is from the public registration APIs provided by the sensor framework. If it is, we allow this IPC because we have already hooked the registration APIs. If not, we deny this IPC because we do not know the customized way in which the app is retrieving and handling sensor events. Denial of such attempts should not affect benign apps' access to sensors because it is abnormal, inefficient, and error-prone to directly access system services through IPC instead of using APIs provided by the system.

Policy Manager and runtime communication Policy Manager is a standalone app running on the same Android device with the instrumented apps, which manages and deploys control policies for all the instrumented apps. At runtime, the wrappers in the instrumented apps need to get the most up-to-date control policies from Policy Manager to decide whether the app is allowed to access sensors. Different from prior works [17, 18] that establish IPC between the app and the manager every time an API call is hooked, Policy Manager only establishes one IPC to an instrumented app when there is a change in the app's control policy. This is a consideration of runtime overhead because the frequency of sensor events is quite high (at least 5 Hz). In Sensor Guardian, every app is instrumented with an internal light-weight Policy Manager that handles IPC with the external Policy Manager and caches current policies only for this app. When a sensor event is hooked, there is no need to ask the external Policy Manager for control policies through IPC. The hooks only need to get the cached policies from the internal Policy Manager in the same process, which has lower overhead than IPC. Though this strategy may delay the new control policy to take effect, the latency is quite low, which is the time of only one IPC process.

Glue together Figure 2 shows the whole process of Sensor Guardian's static instrumentation and runtime control. In preparation, we implemented aforementioned API wrappers both in Java and native. The Java wrappers and the internal Policy Manager, say $\texttt{manager}_{\texttt{int}}$, are compiled into a dex file called $\texttt{dex}_{\texttt{prep}}$. The native wrappers and the code to hook native API calls are compiled into a native shared library, say $\texttt{lib}_{\texttt{prep}}$. Note that $\texttt{dex}_{\texttt{prep}}$ and $\texttt{lib}_{\texttt{prep}}$ only need to be generated once for all apps because these preparations do not depend on any details about a specific app. As a result, the time to generate these

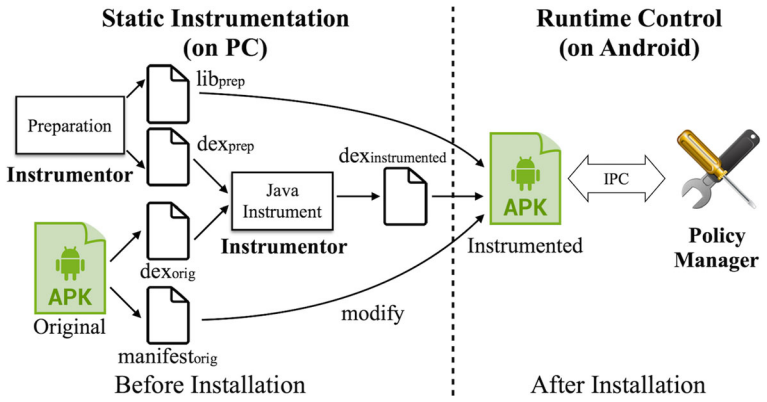

Fig. 2 Whole process of Sensor Guardian. The whole process of Sensor Guardian's static instrumentation and runtime control

two preparations is not included in the measurement of instrumentation time in Section 4.1

Instrumentation starts when Instrumentor extracts essential files from the original APK file, including the manifest file ($\texttt{manifest}_{\texttt{orig}}$) and dex file ($\texttt{dex}_{\texttt{orig}}$). Java Instrumentation takes $\texttt{dex}_{\texttt{prep}}$, $\texttt{manifest}_{\texttt{orig}}$, and $\texttt{dex}_{\texttt{orig}}$ as its input and generates an instrumented dex file ($\texttt{dex}_{\texttt{instrumented}}$) as output. In order to ensure that $\texttt{manager}_{\texttt{int}}$ can get the available policies at the start of an app, we implemented $\texttt{manager}_{\texttt{int}}$ as a subclass of `android.app.Application` and modified the `application` tag in $\texttt{manifest}_{\texttt{orig}}$ to set $\texttt{manager}_{\texttt{int}}$ as the first class to be instantiated. In the case that $\texttt{manifest}_{\texttt{orig}}$ already sets an application class, say $\texttt{class}_a$, we change the superclass of $\texttt{manager}_{\texttt{int}}$ to $\texttt{class}_a$ and instantiate the superclass after $\texttt{manager}_{\texttt{int}}$'s instantiation. We also load $\texttt{lib}_{\texttt{prep}}$ in $\texttt{manager}_{\texttt{int}}$ at the app's start to establish Native Instrumentation.

At last, we repackage $\texttt{dex}_{\texttt{instrumented}}$, $\texttt{lib}_{\texttt{prep}}$, and the modified manifest file into the original APK file to generate the instrumented APK file. And we sign the instrumented APK file with a new certificate. When an APK file is modified and repackaged, its signature is inevitably destroyed and we cannot recover its original signing key. This is a problem in all static instrumentation techniques [16–18]. We adopt the method in [18], in which we generate a new certificate for each app we have encountered and keep the relationship between the certificate and the app. For those apps developed by the same developer, we use the same certificate. In this way, we maintain the functionality of signature that works as a proof of authorship in Android.

3.3 Control policy

We implement several simple yet effective control policies in Sensor Guardian. All these policies take effect by modifying the data in the sensor events that Sensor Guardian intercepts, instead of changing apps' work flow. Here, we elaborate the policies that we support. Note that the supported policies could be expanded by security experts in the future.

Allow All or Deny All The Allow All policy does not set restrictions on the app to access any sensor. In the opposite, Deny All denies all the access attempts of an app on all the sensors. When implementing the Deny All policy, we replace the real data in the sensor events with fixed and unreasonable data (e.g., set the value of step counter as always −1).

Randomization Randomization here means randomizing sensor data for apps to make them unable to understand and infer the real status and environment of the phone. We enforce this policy by replacing the real sensor data with random data generated by Java Random [26]. In this way, the sensor data retrieved by the app is determined by the randomization method instead of the real hardware. As a result, it cannot reflect real device movement or ambient environment, which cannot be used to infer the user's behavior pattern and privacy.

User Select User Select is not a specific policy that decides the way for apps to access sensors. It is a management strategy that Sensor Guardian allows the user to decide which app can access what sensors in what ways by providing her an interactive interface for selection (Fig. 3). This policy is combined with the above policies, that is, the user can decide whether to allow, deny, or randomize an app's access to a specific sensor. For example, the user can deny app_a to access accelerometer while randomize app_b's access to gyroscope.

The motivation for User Select policy is to provide users with the capability of controlling apps' behavior by their own. This user-driven policy and the interactive interface

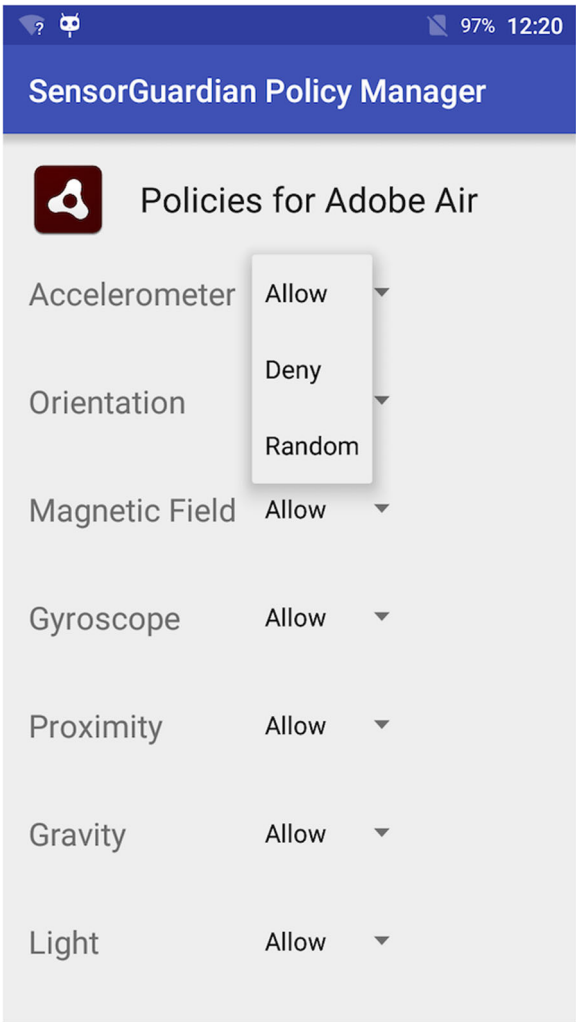

Fig. 3 Screenshot of Policy Manager. A screenshot of Policy Manager's interactive interface. Sensor Guardian allows the user to decide which app can access what sensors in what ways by providing her this interactive interface for selection

of Policy Manager are mostly learnt from similar functionalities in the system settings of iOS and Android 6.0, that is, users can decide whether apps' access to system resources are allowed by switching on or off in the system settings. For example, after Android 6.0, users can turn on or off the Contacts permission for an app in the system Settings app. The main difference here is that sensors are not as self-explained as other system resources, e.g., Contacts capabilities allow apps to access contacts, while gyroscope does not indicate what it can be used for. In addition, the purposes for apps to use a specific sensor are not obvious, e.g., apps can use gyroscope for both shaking detection and keystroke inference. As a result, users' selection on the policies of controlling apps' access to sensors may affect their normal functionalities. In order to reduce the effect on apps' normal functionalities, Sensor Guardian now shows a warning on the status bar (or makes a toast for those apps running in fullscreen) when an app requires access to the sensors that are being controlled by `Deny` or `Randomization` policies. The warning indicates what sensor access of this app is being controlled. With such warnings, users are able to know whether it is Sensor Guardian's control that affects the app's normal functionalities and, if true, make changes to the corresponding control policies in Policy Manager.

When there is a need to deploy a new kind of control policy, it only requires the policy maker to extend the internal Policy Manager before instrumentation, which does not affect any other parts of Sensor Guardian.

3.4 Deployment strategy

Sensor Guardian can be used in several ways. First, it can be used by a normal user who concerns about her privacy. The user can use Sensor Guardian to automatically instrument the apps that may access her sensor data and install the instrumented ones and Policy Manager. In Policy Manager, she can use the interactive interface provided by Policy Manager to control apps' access on sensors.

Second, Sensor Guardian can be used by Android app markets to provide privacy protection enhancement on apps. In this way, apps can be instrumented by Instrumentor on the market server after they are submitted by developers to the market and before the market publishes them to users. Policy Manager can be published as a standalone app, which can be downloaded by users manually or by the instrumented apps automatically.

4 Evaluation

In this section, we evaluate Sensor Guardian in several aspects, including its static instrumentation overhead and runtime control overhead. The Instrumentor runs on a regular PC with eight processors of Intel Core i7-4770 CPU, 3.40 GHz, 16 GB memory, and Linux kernel 3.8.0. We crawled 2200 top free apps from all 28 categories on Google Play Store [27]. By disassembling them and grepping sensor-related APIs in their Java and native assembly code files, we find that 719 apps are using sensor-related APIs and select these apps as our evaluation set. We instrument them and run their instrumented versions on a Samsung GT-I9260 smartphone with Android 5.1.1, to examine the efficiency and effectiveness of Sensor Guardian. Though the evaluation set is relatively small compared to the total number of Android apps on Google Play, this set covers all categories on Google Play and is enough to evaluate Sensor Guardian's performance.

4.1 Instrumentation overhead

Sensor Guardian can successfully instrument all the apps in the evaluation set. Here, we examine the overhead of

Sensor Guardian's instrumentation including its instrumentation time and impact on app size.

Instrumentation time During instrumentation, we record the start time (in seconds) of instrumenting an app and the end time when the instrumentation finishes. The instrumentation starts when Instrumentor extracts essential files (e.g. dex_{orig} and $\text{manifest}_{\text{orig}}$) from the app's APK file and ends when the instrumented APK is signed. The instrumentation time is measured by calculating the difference between the start and end time. Note that the preparations of generating $\texttt{dex}_{\texttt{prep}}$ and $\texttt{lib}_{\texttt{prep}}$ are not included in the measurement of instrumentation time since Instrumentor only needs to generate them for once and these preparations are completed in a short time before all apps' instrumentation. Before each app's instrumentation, we clear the RAM and swap caches in order to eliminate caching effects.

All apps are instrumented in 20 s. The average instrumentation time is 8 s. Figure 4 shows the number of apps instrumented in different time, which reveals that most apps are instrumented in 7, 8, and 9 s. As noted above, the instrumentation time for an app includes the time of extracting its APK file, modifying dex file and manifest file, recompression and signing, but not the time for generating $\texttt{dex}_{\texttt{prep}}$ and $\texttt{lib}_{\texttt{prep}}$. The result indicates that Sensor Guardian does not cost too much time during static instrumentation.

Impact on app size We also measure the size increase of apps to show the impact of Sensor Guardian's instrumentation. Sensor Guardian incurs size increase in both dex file and native libraries. Dex file's size increase is mainly caused by the inserted hooks and internal Policy Manager $\texttt{manager}_{\texttt{int}}$. Native libraries' size increase is caused by adding the extra library $\texttt{lib}_{\texttt{prep}}$. The $\texttt{lib}_{\texttt{prep}}$ has a fixed size of 22 KiB, while the size increase of dex files vary between apps. Figure 5a, b shows the cumulative distribution function (CDF) for the size increase of dex files and APK files. In fact, the average size increase of dex files and APK files are 0.5 and 0.6%, respectively. All these results show that Sensor Guardian's instrumentation has negligible impact on apps' size.

Summary Both the time and size overhead incurred by Sensor Guardian's instrumentation is negligible, especially considering the fact that it instruments both the Java and native code of apps.

4.2 Runtime overhead

After instrumentation, we manually run those instrumented apps on the experiment device. No app crashed under all policies, i.e., Allow All, Deny All, and Randomization, which means that Sensor Guardian's instrumentation does not affect apps' normal execution. In fact, as suggested in Section 3.3, all these policies take effects by modifying the data in intercepted sensor events, instead of stopping apps from executing event listener callbacks (for Java API) or polling the sensor event queue (for native API). Even under Deny All and Randomization policies, Sensor Guardian just replaces the data in sensor events with fixed and unreasonable data (Deny All) or randomized data (Randomization). As a result, Sensor Guardian's control should not interrupt or stop apps' execution.

At runtime, the instrumented apps' access to sensors are controlled by Sensor Guardian. This control may incur overhead to these access attempts. Here, we examine the runtime overhead incurred by Sensor Guardian's protection. The third-party apps we crawled from app market are not suitable for observing this overhead because they do not always use sensors, most of them only rely on sensor data occasionally. We need benchmark apps that intensively retrieve sensor data to better observe the overhead incurred by Sensor Guardian's control.

Time overhead Each time a sensor event emerges, Sensor Guardian checks the current control policy and decides how to control the intercepted sensor events. This check and control decision may incur extra time overhead that delays the instrumented app to get sensor events

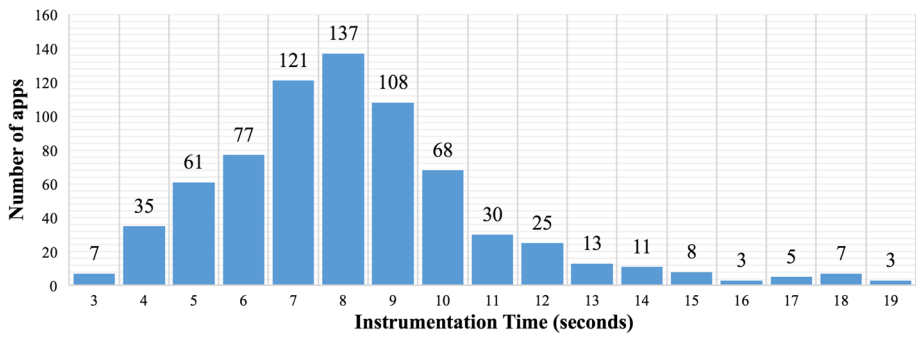

Fig. 4 Instrumentation time. The number of apps instrumented in different time. Most apps are instrumented in 7, 8, and 9 s

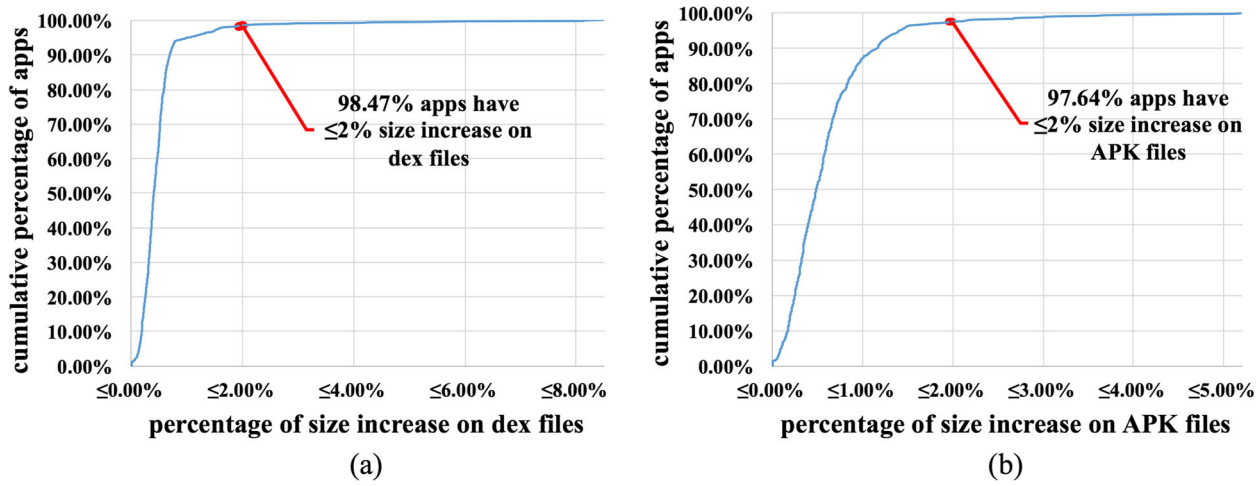

Fig. 5 a, **b** CDF for the size increase of dex and APK files. The cumulative distribution function (CDF) for the size increase of dex files and APK files incurred by Sensor Guardian's instrumentation

and data. With Sensor Guardian's control, the time for apps to receive sensor event is decided by its sampling rate and the extra time overhead incurred by the check and control. We developed a microbenchmark to measure how long Sensor Guardian's control delays the instrumented app to get sensor data. This microbenchmark retrieves data from different sensors at different sampling rates. In the microbechmark, we record the timestamp (in nanoseconds) of receiving each sensor event and calculate the time interval between each two continuous sensor events' timestamps. We measure the mean time interval of 30,000 sensor events for both the original and the instrumented microbenchmarks and name them as int_{orig} and int_{inst}, respectively. Then, we measure the extra time overhead incurred by Sensor Guardian's control by calculating $int_{inst} - int_{orig}$. We evaluate the time overhead for different sensor types at different sampling rates. Android provides several sampling rates for accessing sensor data. We choose the highest two, SENSOR_DELAY_FASTEST (as fast as possible, accurate rate depends on devices) and SENSOR_DELAY_GAME (50 Hz). Table 2 shows the time overhead for different sensors at different sampling

Table 2 Time overhead for different sensors at different sampling rate and under different policies

Sensor type	Sampling rates	Original mean time interval between events (int_{orig}, unit: μs)	Policies	Absolute time overhead ($int_{inst} - int_{orig}$, unit: μs)	Relative time overhead (($int_{inst} - int_{orig}$)/int_{orig}, unit: ‰)
Accelerometer	Fastest	8392.208918	Allow All	0.07	0.08
			Deny All	0.001	0.001
			Randomization	0.02	0.02
	Game	20141.28197	Allow All	0.08	0.04
			Deny All	0.03	0.01
			Randomization	0.16	0.08
Magnetic field	Fastest	16784.63194	Allow All	0.008	0.005
			Deny All	0.009	0.005
			Randomization	0.002	0.001
	Game	20141.59088	Allow All	0.02	0.009
			Deny All	0.004	0.002
			Randomization	0.04	0.02
Gyroscope	Fastest	5035.314463	Allow All	0.003	0.001
			Deny All	0.01	0.02
			Randomization	0.01	0.01
	Game	20141.28592	Allow All	0.16	0.08
			Deny All	0.12	0.06
			Randomization	0.03	0.02

rates and under different policies. The absolute time overhead is calculated by $\text{int}_{\text{inst}} - \text{int}_{\text{orig}}$ (as mentioned above), and the relative time overhead is calculated by $(\text{int}_{\text{inst}} - \text{int}_{\text{orig}})/\text{int}_{\text{orig}}$. In fact, int_{orig} is decided by the sampling rate. Table 2 reveals that the highest absolute time overhead is 0.16 μs, which is much less than 20 μs in [6], while the highest relative time overhead is also negligible as 0.08 $^{0}/_{000}$.

CPU and memory overhead We use the benchmark app Ambulation [4] to examine the CPU and memory overhead of Sensor Guardian's protection. Ambulation continuously requests sensor data, which is suitable to examine the CPU and memory overhead. It was also used in the prior work ipShield [6] to evaluate performance. Different from ipShield that sets Ambulation's sampling rate at 1 Hz during evaluation, we set the sampling rate higher than 1 Hz because the CPU usage cannot be observed on our experiment device when the sampling rate is set to 1 Hz. Instead, we choose SENSOR_DELAY_NORMAL (5 Hz), SENSOR_DELAY_GAME (50 Hz), and SENSOR_DELAY_FASTEST.

Table 3 shows the CPU and memory overhead of Sensor Guardian's protection at different sampling rates and under different policies. It reveals that, maximally, Sensor Guardian incurs 0.17% CPU overhead and 0.1 MB memory overhead, which are less than those (3% CPU and 0.5 MB memory) in ipShield [6].

Power consumption Low power consumption is essential for all mobile apps, especially for apps using sensor data because frequently accessing sensors would drain the battery. As a result, a protection system targeted at sensors should not incur too much extra power consumption. We measure the power consumption overhead in the same way as in [28] and [6], that is, we measure how long it takes for the original and instrumented apps to consume the same amount of power (percentage of battery). This measurement is based on the fact that the more power an app consumes, the faster it drains the battery. We still use Ambulation as the benchmark app and choose the sampling rates as SENSOR_DELAY_FASTEST. During its execution, we turned off all network interfaces and turned down the screen's brightness to the lowest. We measure how long it takes for the app to drain the battery from 100 to 90%. Table 4 shows the time cost under different policies. By comparing the time to drain battery for the original benchmark and the one under different control policies, we can evaluate the power consumption overhead incurred by Sensor Guardian. Table 4 illustrates that the highest power consumption overhead incurred by Sensor Guardian is 2.6%. This overhead is much less than prior works as 7.6% [28] and 8.2% [6].

Summary Sensor Guardian has much improved performance compared to prior works [6, 28] during runtime control in all aspects as time, CPU, memory, and power consumption. This performance improvement is mainly because of its lazy strategy on runtime communication that the instrumented app only needs to query the intra-process $\texttt{manager}_{\texttt{int}}$ instead of establishing an expensive IPC with the external Policy Manager (see Section 3.2). The low runtime overhead makes Sensor Guardian practical to solve the PIS problem.

4.3 Protection effectiveness

Under Sensor Guardian's control, all apps cannot arbitrarily retrieve sensor data. We run the apps in our evaluation set under Sensor Guardian's Deny All and Randomization control policies to evaluate the protection's effectiveness. All apps' runtime behaviors are affected. We also examine the effectiveness of Sensor Guardian's control on the apps that really infer the user's sensitive information based on sensor data, i.e., the malicious app that launches PIS attack. In fact, the benchmark app Ambulation [4] that we used in runtime overhead evaluation launches a PIS attack, in which it continuously retrieves sensor data from accelerometer to infer the user's transportation state, like still, walking, or running. We run Ambulation under Sensor Guardian's Deny All and Randomization control

Table 3 CPU and memory overhead at different sampling rate and under different policies

Sampling rates	Policies	CPU overhead (%)	Mem overhead (MB)
Fastest	Allow All	0.1	0.03
	Deny All	0.16	0.005
	Randomization	0.01	0.008
Game	Allow All	0.06	0.07
	Deny All	0.17	0.01
	Randomization	0.02	0.04
Normal	Allow All	0.13	0.05
	Deny All	0.06	0.1
	Randomization	0.17	0.07

Table 4 Time to drain the device battery from 100 to 90% under different policies

	Original	Allow All	Deny All	Randomization
Time to drain battery from 100 to 90% (T)	36′ 45″ (T_O)	35′ 57″	35′ 48″	35′ 48″
$T - T_O$	0	48″	57″	57″
Power consumption overhead $(T - T_O)/T_O$	0	2.1%	2.6%	2.6%

policies. It cannot infer the right transportation state. As shown in Fig. 6, the phone is placed still on the desk. But under the Randomization policy, Ambulation's inferred state is running.

4.4 Findings

Among all the 719 apps in our evaluation set, 648 (90%) use only Java APIs to access sensors, 22 (3%) use only native APIs, and 49 (7%) use both Java and native APIs.

During static instrumentation, we manually looked into the assembly code of the apps that use sensor-related Java APIs and checked the types of sensors passed to the registration methods, in order to understand what sensors may be used in these apps. Though the collected results may not be complete, we can have a rough understanding of the usage of sensors in Android apps.

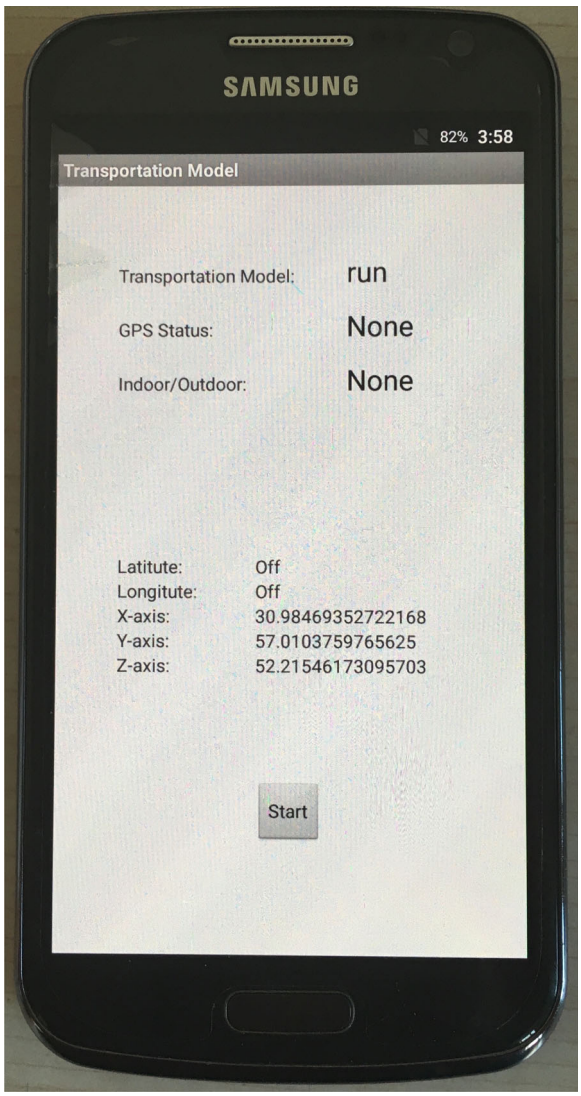

Fig. 6 Ambulation infers wrong transportation mode under Sensor Guardian's control. Under the Randomization policy, Ambulation infers the transportation mode to be running though the device is placed still on the desk

Figure 7 shows the number of apps categorized by the number of sensors they are using. It reveals that 284 (39.5%) apps use only one sensor and 248 (34.5%) use two sensors, which are the majority (74%) in the evaluation set. The result suggests that most apps only require a small number of sensors.

Besides the number of sensors being used by apps, we also examine the types of sensors they use. Table 5 shows the popularity of different types of sensors among apps, which reveals that accelerometer is the most popular sensor. It suggests that 610 (84.8%) apps are using accelerometer, followed by magnetic field (31.6%), proximity (28.1%), and gyroscope (14.9%). We believe that the reason for accelerometer to be the most popular is that it can be used to detect a lot of device status, like shaking, rotation, and vibration. Actually, among the 284 apps that only use one sensor, we find that 201 (70.8%) apps use and only use accelerometer.

During evaluation, we manually examine the assembly code of some apps that use accelerometer to understand why it is so popular. We find that a common usage of accelerometer in apps, like Booking and Tumblr, is to detect whether the user is shaking the device. To detect this high-level motion event, Android developers need to implement the algorithm based on raw sensor data by themselves. Unlike Android, iOS does not only provide raw sensor data to developers but also supports APIs to detect high-level motion events, like device shaking, in order to lighten developers' burden. This strategy also reduces the risks of exposing users' privacy because, for apps that reply on accelerometer only for detecting device shaking, they do not need to access raw data anymore. Instead, they only need to ask the system whether the device is shaking through the high-level APIs. We recommend Android to provide such high-level motion event APIs, which could also be applied to other sensors and their high-level motion events. These high-level APIs can also cooperate with the permission system, e.g., apps with permission.accelerometer_fine could retrieve raw sensor data while those with permission.accelerometer_coarse can only use high-level APIs.

We also extract common combinations of some popular sensors, which is demonstrated in Table 6. It reveals that the combination of magnetic field and accelerometer is the most popular sensor combination among apps. We further categorize the apps in the evaluation set based on their descriptions on the app market and check whether apps in the same category use the same sensors, in order to characterize why apps are using sensors. Table 7 demonstrates some category examples, for example, nearly all Map apps need to use accelerometer during navigation.

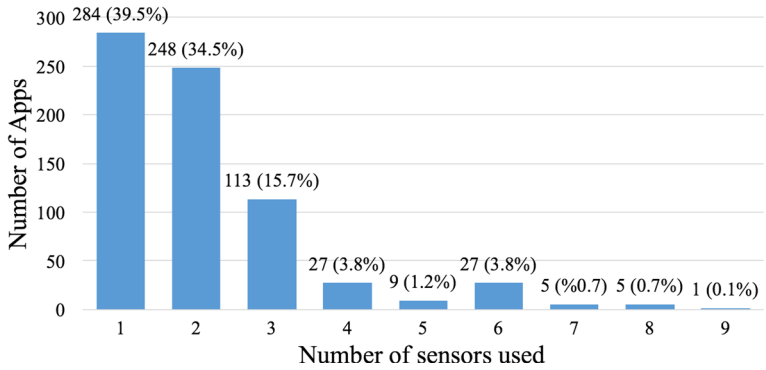

Fig. 7 Number of apps using different number of sensors; 284 (39.5%) apps use only one sensor and 248 (34.5%) use two sensors, which are the majority (74%) of the apps in the evaluation set

5 Discussion

In this section, we discuss Sensor Guardian's advantages and disadvantages, including the completeness of its API hooking, the incurred attack surfaces to apps, Sensor Guardian's strengths and weaknesses, and recommendations for sensor management on Android.

5.1 Completeness of hooking

In this section, we elaborate the completeness of Sensor Guardian's hooking by examining all the possible ways in which apps can access sensors on Android.

1. Android is based on the Linux kernel. Hardware devices, including sensors, are managed by the Linux kernel. The kernel does not provide corresponding system calls to directly access sensors. It organizes sensor devices as device files. Android restricts that only the SensorService system service can access the sensor device files by setting read-write permissions and SEAndroid policies on these files. As a result, the only way for user-mode apps to access sensors is to communicate with the SensorService through Binder IPC.
2. In Sensor Guardian, we hook `ioctl()` in libc.so, the main function through which all IPCs are sent, and examine apps' attempts on accessing sensors. In this way, all access to sensors should be hooked and controlled. But a malicious app can bypass such hooks by implementing their own `ioctl()` function instead of using the monitored one. Actually, the `ioctl()` function is a wrapper of the `ioctl()` system call. One possible solution that we are working on is to examine the instructions of calling this system call in all the native binaries loaded by the app and replace them with calls to a new wrapper of `ioctl()` system call.
3. We hook all the public Java and native APIs for registering sensor events and examine the call stack trace in `ioctl()`. The examination in `ioctl()` makes sure that the allowed access to the SensorService is from hooked public APIs. By this means, apps cannot use internal or hidden APIs to bypass Sensor Guardian's hooking.

We make Sensor Guardian's protection as complete as possible by handling all the possible situations mentioned above. But it may still miss some conditions in which

Table 5 Popularity of different sensors

Sensor type	Number of apps using the sensor	Percentage of apps using the sensor (%)
Accelerometer	610	84.8
Magnetic field	227	31.6
Proximity	202	28.1
Gyroscope	107	14.9
Orientation	96	13.4
Light	74	10.3
Rotation vector	60	8.3
Gravity	42	5.8
Linear accelerometer	41	5.7
Pressure	25	3.5
Other sensors	34	4.7

Table 6 Number of apps using different combinations of sensors

Sensor combination	Number of apps
Magnetic field, accelerometer	94
Proximity, accelerometer	62
Magnetic field, proximity, accelerometer	28
Gravity, linear accelerometer, rotation vector, Accelerometer, magnetic field, gyroscope	21
Orientation, accelerometer	20
Light, proximity	12

Table 7 Some apps in the same category use the same sensors

Category	Same sensors used	App example
Map	Accelerometer	BaiduMap, Maps.Me, Minimap
Health	Step detector, accelerometer	Mi Fit, FitbitMobile
First-person game	Linear accelerometer, rotation vector, Accelerometer, gyroscope	3D Maze 2, Squadron, Sniper SWAT

apps use some rare methods to retrieve sensor data in customized ways (like the abovementioned implementing their own `ioctl()`) or dynamically change their behaviors. These unhandled situations are discussed in Section 5.4.

5.2 Attack surfaces incurred by Sensor Guardian

After static instrumentation, Sensor Guardian adds new code into the instrumented app. If the inserted code is not handled properly, it may introduce new attack surfaces and cause new security problems to the app. Here, we discuss those potential attack surfaces and our mitigation.

The instrumentation does not modify the original control flow of an app. It only replaces the targets of those calls to sensor-related API. The only potential attack surface incurred by the inserted code is that it introduce a new IPC channel to the original app. This IPC channel is used by the internal Policy Manager to get up-to-date control policies from external Policy Manager. Without considering potential bugs in Android's IPC mechanism, the only possible way for a third-party app to attack the instrumented app through this IPC channel is to impersonate the external Policy Manager to the internal Policy Manager. But, with the help of Android IPC mechanism and PackageManager, the internal Policy Manager can verify the package name and signature of the connected external Policy Manager, which prevents the impersonation attack.

5.3 Strengths of Sensor Guardian

Besides the completeness of hooking and the incurred attack surfaces to Android apps, in this section and Section 5.4, we also discuss the strengths and weaknesses of Sensor Guardian, in order to have a full understanding about Sensor Guardian's protection. The strengths of Sensor Guardian are listed as below.

First, Sensor Guardian has negligible overhead. Section 4.2 shows that Sensor Guardian has negligible overhead on time, CPU, memory, and power consumption, during runtime control. All the overhead is much improved compared to prior works [6, 28]. Also, Sensor Guardian has low overhead during static instrumentation, that is, it can instrument an app in a few seconds while incurring small size increase to the app. The low overhead in both static instrumentation and runtime control makes Sensor Guardian practical.

Second, Sensor Guardian is not affected by the problem of Android fragmentation [18, 29]. This problem arises when a variety of customized Android versions are used in different vendors' devices. This problem makes it difficult to provide a universal patch to enforce new security enhancement at the system level on all Android devices, due to misaligned incentives from different parties including Google, hardware vendors, and cell phone carriers [18, 30, 31]. Sensor Guardian does not need to modify the system to deploy the security enforcement, thus not affected by the Android fragmentation problem. As long as the original app can run on a device, its instrumented version can also run on that device under Sensor Guardian's control. Thus, Sensor Guardian can be used to handle the PIS problem before Android notices the problem and starts to enforce new access control on those unprivileged sensors at the system level, such as extending existing permission model or deploying other control systems like [6]. Even Sensor Guardian can be used now to handle the new scenarios in the PIS problem, like inferring user privacy by exploiting smartwatches [7, 10].

5.4 Weaknesses and limitations

Sensor Guardian employs the method of static instrumentation. This method is not a silver bullet for all situations and may face several problems that are inevitably difficult to deal with. To the best of our knowledge, we have not found existing work that completely solves these problems in static instrumentation. Here, we elaborate the weaknesses of this method and our future work of mitigating these weaknesses.

1. Apps can detect whether they are modified or repackaged by examining the signature or signing identity of their APK files. An app may refuse to run to avoid being controlled when this repackaging is detected. In the future, we will address this problem by hooking apps' checks on their signatures, in order to hide the traces of instrumentation.
2. Apps may dynamically change their behavior at runtime. We try to solve this weakness by hooking critical reflection function and functions like `dlsym()`, but a malicious app could change its behavior in other ways, like through direct jumping. The completeness of Sensor Guardian's hooking is affected when apps dynamically change their behaviors with such unhandled APIs or instructions. We are working to make a full list of all APIs that can be used to dynamically change apps' behaviors. And in the future work, we will not only hook APIs but also hook instructions that change control flow.

3. Apps may load code dynamically in several ways [32]. Though we handled dynamically loaded native shared libraries, it is still not a complete solution. Apps may download new dex files or binaries from network and load code from these downloaded files. Such runtime behaviors may affect the completeness of Sensor Guardian's hooking since static instrumentation techniques cannot predict the files to be downloaded and instrument the dynamically downloaded files at runtime. In the future, we will address this problem by analyzing and hooking apps' dynamically loading behaviors with both static and dynamic methods.

Basically, these weaknesses are all caused by apps' dynamic behaviors at runtime. These runtime behaviors are difficult to be analyzed statically from apps' code (like what Sensor Guardian does now). In the future work, we are planning to combine static and dynamic analysis techniques to make more complete control on these runtime behaviors.

Another limitation in Sensor Guardian is that, currently, we only implemented several simple control policies in Sensor Guardian. Though these policies are effective, they may be inflexible and unintelligent, which may incur false positive when controlling some legitimate apps. In the future, we are going to propose and support other flexible and intelligent polices in Sensor Guardian. For example, User Select policy is still simple and rough. Users may not make correct and precise control on apps' access to sensors. We are working to give users correct recommendations and guides on how to decide control policies for protecting their privacy. Also, we are working to automatically analyze how apps use sensors and distinguish between malicious behaviors and normal functionalities. All these works require analysis on the semantics of apps, especially on the computation that sensor data takes part in. In the future, we will add such semantics analysis to Sensor Guardian.

Currently, Sensor Guardian's instrumentation is implemented on a regular PC. That means, if it is used by a user, the user needs to install the instrumented app from PC, which may be inconvenient for an ordinary user. In the future work, we are planning to port Sensor Guardian's instrumentation to Android devices to avoid such inconvenience.

5.5 Recommendations for Android

Sensor Guardian is not a complete solution for the PIS problem on Android. It is a potential mitigation before Android pays attention to and takes action to the PIS problem. It is more complete and strict to enforce sensor control policies at the system level. One possible solution is to set permissions on those unprivileged sensors and prevent apps from reading sensor data in the background. Another possible solution is that the system can give the user an obvious indication of what sensors are being used and which app is using these sensors. And the system can also enable users to revoke apps's ability of using specific sensors when they are doing sensitive operations, like entering PIN code.

6 Related work

Sensor Guardian is proposed to solve the aforementioned PIS problem. Many techniques have been proposed to solve this problem, while many others present new attacks that use sensors to infer users' privacy. To solve the PIS problem, Sensor Guardian employs and improves static instrumentation techniques. In this section, we will review prior works about the PIS problem and static instrumentation techniques on Android.

PIS problem Several prior works focus on addressing the PIS problem by hooking and controlling Android apps' access to sensors onboard. Cai et al. [33] explores the vulnerability of attackers sniffing sensors on smartphones. TaintDroid [34] modifies the Android system and framework to establish system-wide dynamic taint tracking and analysis for sensitive data including data from sensors. Based on TaintDroid, AppFence [35] substitutes sensitive data demanded by apps with innocuous shadow data to protect users' privacy, which inspired Sensor Guardian's Randomization policy. FlaskDroid [36] modifies the Android system to enforce mandatory access control on both Android's middleware and kernel layers, including on the sensor data. ipShield [6, 37] tries to solve the PIS problem by modifying the Android system and monitoring apps' access to innocuous sensors. It then uses the monitored information to provide the user with privacy risk assessment. SemaDroid [38] also modifies the Android system to extend its sensor management framework and enforces several fine-grained control policies. Perceptual Assistant (PA) [28] follows ipShield to allow users to customize control policies. It relies on Xposed [39] on rooted Android devices to dynamically hook apps' related API calls. SSG [40] hooks apps' sensor-related APIs by modifying the Android system, and it needs to be installed with root privilege. Compared to these prior works, Sensor Guardian has much improved performance and incurs less overhead during runtime control. And those methods relying on modifying the system may be affected by the problem of Android fragmentation, while Sensor Guardian is not.

Some other works focus on providing privacy protection when users share their sensor data with remote data consumers. Mun et al. [41] proposes Personal Data Vaults (PDVs) to help users control and filter their sensor data shared with remote content-service providers. Choi et al. [42] also proposes an architecture, called SensorSafe, for

users to share their sensor data with remote data consumers in a privacy-preserving way. AnonySense [43] helps users to anonymize their sensor data when sharing among multiple users. Zhang et al. [44] proposes a privacy-preserving technique for smartphone sensing data aggregation. Different from these works, Sensor Guardian directly controls local apps' access on sensors no matter whether they share the sensor data with remote data consumers.

Many works try to infer different kinds of privacy or information based on the innocuous sensors in a direct way [4, 7, 10] or as a side channel [1–3, 8, 45, 46]. Kwapisz et al. [45] uses built-in accelerometers to recognize users' physical activity. Schlegel et al. [47] leverages microphone to steal users' sensitive privacy during phone call. Zhou et al. [48] collects audio status from a GPS navigator to fingerprint a driving route. Mylonas et al. [5] uses sensors on smartphones in digital forensics as either direct evidence or indirect evidence. Shen et al. [49] proposes a sensing stack on Internet of Things (IoT) devices. Reddy et al. [4] uses GPS and motion sensors on the smartphone to infer the user's transportation modes. Yang et al. [50] infers sensors in a building to make undesired occupancy-related inferences. Dey et al. [51] can identify each unique smartphone by leveraging the imperfections of their built-in accelerometers as fingerprints. Miluzzo et al. [52] uses motion sensors to infer the location of users' taps on the screen. More stealthily, [9, 53–56] use unrestricted sensors, including gyroscope, accelerometer, magnetometer, and barometer to infer users' location, driving patterns, and traveled routes. Xu et al., Cai and Chen, Shen et al., Aviv et al., and Owusu et al. [1, 2, 8, 46, 57] use motion sensors to infer users' keystroke on the smartphone's screen because typing on different locations on the screen may cause changes on the device's motion status, which is reflected on the motion sensors. Similarly, [3] uses accelerometer to infer users' keystroke. However, the keystroke is not on the smartphone's screen, it is on another physical keyboard that is placed near the smartphone on the same desk. Hussain et al. [58] conducts a survey of existing attacks that use motion sensors for keylogging on smartphones. Inspired by these keylogging techniques, [59] conducted a study to understand users' concerns on data collected by wearable sensors. Song et al. [60] suggests that these keylogging attacks can be addressed by reducing sensor data accuracy and generating random keyboard layout. Liu et al. [7] and Wang et al. [10] look into a new situation that users may type on a keypad or keyboard with a smartwatch worn on her wrist. As the motion sensor on the smartwatch can track the movement of the device, it can also track the movement of the user's hand, which makes it possible to infer the keys she strokes on the keypad or keyboard.

Static instrumentation on Android Static instrumentation on Android apps can be divided into two categories as Java and Native Instrumentation. Java Instrumentation is also called bytecode rewriting, which instruments on apps' Dalvik bytecode. It uses static analysis to identify sensitive API calls and modifies the APK file to hook these calls. Jeon et al. [17] uses bytecode rewriting to enforce finer-grained permissions as an improvement to Android's coarse-grained permission system. RetroSkeleton [16] is a rewriting framework for customizing Android apps' behaviors, which supports several policies. These bytecode rewriting techniques do not handle native code in Android apps. Differently, Sensor Guardian does not only rewrite bytecode but also instruments native code. Another difference is that Sensor Guardian does not rely on disassembling tools, instead it directly instrument at Dalvik bytecode level. Hao et al. [19] systematically evaluates the effectiveness of API-level access control using bytecode rewriting and gives recommendations on engineering secure bytecode rewriting tools. In Sensor Guardian, we try to overcome some weaknesses mentioned in that work, for example, we hook reflection APIs and prevent apps from accessing system service directly. When instrumenting native code, Sensor Guardian employs the method in Aurasium [18]. Aurasium monitors and controls apps' attempts to access sensitive information and system resources by hooking low-level system APIs at the start of the app. Differently, Sensor Guardian hooks high-level APIs each time a new library is loaded.

7 Conclusions

In this paper, we propose Sensor Guardian, a new system to prevent the problem of privacy inference based on sensors (PIS). Sensor Guardian hooks and controls apps' Java and native API calls of accessing sensors by statically instrumenting their APK files. The evaluation shows that Sensor Guardian can effectively control apps' access to sensors and incur negligible overhead in several aspects. Specifically, for static instrumentation, the average instrumentation time is 8 s, and the average size increase on APK files is only 0.6%. At runtime, the maximum extra time, CPU, memory, and power consumption overhead incurred by Sensor Guardian are 0.16μs, 0.17%, 0.1 MB, and 2.6%, respectively. All these results show that Sensor Guardian has improved performance compared to prior works.

Endnote

[1] Note that this `invoke()` function is different from the `invoke` instruction stated in the steps of hooking Java APIs: this `invoke()` function is a reflection API, while the `invoke` instruction is a Dalvik bytecode instruction.

Acknowledgements
This work was supported by Tsinghua University Initiative Scientific Research Program (2014z09102).

Authors' contributions
XB developed the Sensor Guardian, conducted the experiments for the evaluation, and drafted the manuscript. JY participated in the Sensor Guardian's design and revised the manuscript. YW analyzed the experiment results and wrote the final version of the manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 21 November 2016 Accepted: 15 May 2017
Published online: 08 June 2017

References

1. Z Xu, K Bai, S Zhu, in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors (ACM, New York, 2012), pp. 113–124
2. L Cai, H Chen, in *HotSec*. Touchlogger: inferring keystrokes on touch screen from smartphone motion (USENIX Association, Berkeley, San Francisco, 2011)
3. P Marquardt, A Verma, H Carter, P Traynor, in *Proceedings of the 18th ACM Conference on Computer and Communications Security*. (sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers (ACM, New York, 2011), pp. 551–562
4. S Reddy, M Mun, J Burke, D Estrin, M Hansen, M Srivastava, Using mobile phones to determine transportation modes. ACM Trans. Sens. Netw. (TOSN). **6**(2), 13 (2010)
5. A Mylonas, V Meletiadis, L Mitrou, D Gritzalis, Smartphone sensor data as digital evidence. Comput. Secur. **38**, 51–75 (2013)
6. S Chakraborty, C Shen, KR Raghavan, Y Shoukry, M Millar, M Srivastava, in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. ipshield: a framework for enforcing context-aware privacy (USENIX Association, Seattle, 2014), pp. 143–156
7. X Liu, Z Zhou, W Diao, Z Li, K Zhang, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. When good becomes evil: keystroke inference with smartwatch (ACM, New York, 2015), pp. 1273–1285
8. C Shen, S Pei, Z Yang, X Guan, Input extraction via motion-sensor behavior analysis on smartphones. Comput. Secur. **53**, 143–155 (2015)
9. S Narain, TD Vo-Huu, K Block, G Noubir, in *IEEE Symposium on Security and Privacy*. Inferring user routes and locations using zero-permission mobile sensors. (IEEE, 2016)
10. C Wang, X Guo, Y Wang, Y Chen, B Liu, in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. Friend or foe?: your wearable devices reveal your personal pin. (ACM, 2016), pp. 189–200
11. Android Sensor. https://developer.android.com/reference/android/hardware/Sensor.html. Accessed 18 Nov 2016
12. iOS Sensor. https://developer.apple.com/reference/coremotion. Accessed 18 Nov 2016
13. Windows Sensor. https://msdn.microsoft.com/windows/uwp/devices-sensors/sensors. Accessed 18 Nov 2016
14. Sensors Overview. http://developer.android.com/guide/topics/sensors/sensors_overview.html. Accessed 18 Nov 2016
15. Android Binder. https://developer.android.com/reference/android/os/Binder.html. Accessed 18 Nov 2016
16. B Davis, H Chen, in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. Retroskeleton: retrofitting Android apps. (ACM, New York, 2013), pp. 181–192
17. J Jeon, KK Micinski, JA Vaughan, A Fogel, N Reddy, JS Foster, T Millstein, in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. Dr. Android and Mr. Hide: fine-grained permissions in android applications. (ACM, New York, 2012), pp. 3–14
18. R Xu, H Saïdi, R Anderson, in *USENIX Security Symposium*. Aurasium: practical policy enforcement for android applications, (Bellevue, 2012), pp. 539–552
19. H Hao, V Singh, W Du, in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. On the effectiveness of api-level access control using bytecode rewriting in android. (ACM, New York, 2013), pp. 25–36
20. smali. https://github.com/JesusFreke/smali. Accessed 18 Nov 2016
21. apktool. http://ibotpeaches.github.io/Apktool/. Accessed 18 Nov 2016
22. dexlib2. https://github.com/JesusFreke/smali/tree/master/dexlib2. Accessed 18 Nov 2016
23. Java Reflection. https://docs.oracle.com/javase/tutorial/reflect/member/methodInvocation.html. Accessed 18 Nov 2016
24. Class Method. https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Method.html. Accessed 18 Nov 2016
25. libunity. http://docs.unity3d.com/Manual/android-GettingStarted.html. Accessed 18 Nov 2016
26. Java Random. https://docs.oracle.com/javase/7/docs/api/java/util/Random.html. Accessed 18 Nov 2016
27. Google Play App Store. https://play.google.com/store. Accessed 18 Nov 2016
28. K Zhao, D Zou, H Jin, Z Tian, W Qiang, W Dai, in *Mobile Services (MS), 2015 IEEE International Conference On*. Privacy protection for perceptual applications on smartphones. (IEEE, 2015), pp. 174–181
29. Fragmentation. https://en.wikipedia.org/wiki/Fragmentation_(programming). Accessed 18 Nov 2016
30. X Zhou, Y Lee, N Zhang, M Naveed, X Wang, in *Security and Privacy (SP), 2014 IEEE Symposium On*. The peril of fragmentation: security hazards in android device driver customizations. (IEEE, 2014), pp. 409–423
31. L Xing, X Pan, R Wang, K Yuan, X Wang, in *2014 IEEE Symposium on Security and Privacy*. Upgrading your android, elevating my malware: privilege escalation through mobile os updating. (IEEE, 2014), pp. 393–408
32. S Poeplau, Y Fratantonio, A Bianchi, C Kruegel, G Vigna, in *NDSS*. Execute this! Analyzing unsafe and malicious dynamic code loading in android applications, vol. 14, (2014), pp. 23–26
33. L Cai, S Machiraju, H Chen, in *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*. Defending against sensor-sniffing attacks on mobile phones MobiHeld '09. (ACM, New York, 2009), pp. 31–36. doi:10.1145/1592606.1592614. http://doi.acm.org/10.1145/1592606.1592614
34. W Enck, P Gilbert, S Han, V Tendulkar, B-G Chun, LP Cox, J Jung, P McDaniel, AN Sheth, Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS). **32**(2), 5 (2014)
35. P Hornyack, S Han, J Jung, S Schechter, D Wetherall, in *Proceedings of the 18th ACM Conference on Computer and Communications Security*. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications (ACM, 2011), pp. 639–652
36. S Bugiel, S Heuser, A-R Sadeghi, in *Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies (USENIX, Washington, 2013), pp. 131–146
37. S Chakraborty, KR Raghavan, MP Johnson, MB Srivastava, in *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*. A framework for context-aware privacy of sensor data on mobile systems (ACM, New York, 2013), p. 11
38. Z Xu, S Zhu, in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. Semadroid: a privacy-aware sensor management framework for smartphones. (ACM, New York, 2015), pp. 61–72
39. Xposed. http://repo.xposed.info/module/de.robv.android.xposed.installer. Accessed 18 Nov 2016
40. B Li, Y Zhang, C Lyu, J Li, D Gu, in *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Ssg: sensor security guard for android smartphones (Springer, 2015), pp. 221–233
41. M Mun, S Hao, N Mishra, K Shilton, J Burke, D Estrin, M Hansen, R Govindan, in *Proceedings of the 6th International COnference*. Personal data vaults: a locus of control for personal data streams. (ACM, New York, 2010), p. 17
42. H Choi, S Chakraborty, ZM Charbiwala, MB Srivastava, in *Workshop on Secure Data Management*. Sensorsafe: a framework for privacy-preserving management of personal sensory information. (Springer, Berlin, 2011), pp. 85–100

43. C Cornelius, A Kapadia, D Kotz, D Peebles, M Shin, N Triandopoulos, in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*. Anonysense: privacy-aware people-centric sensing (ACM, New York, 2008), pp. 211–224
44. Y Zhang, Q Chen, S Zhong, Privacy-preserving data aggregation in mobile phone sensing. IEEE Trans Inf. Forensic Secur. **11**(5), 980–992 (2016)
45. JR Kwapisz, GM Weiss, SA Moore, Activity recognition using cell phone accelerometers. SIGKDD Explor. Newsl. **12**(2), 74–82 (2011). doi:10.1145/1964897.1964918
46. AJ Aviv, B Sapp, M Blaze, JM Smith, in *Proceedings of the 28th Annual Computer Security Applications Conference*. Practicality of accelerometer side channels on smartphones. (ACM, 2012), pp. 41–50
47. R Schlegel, K Zhang, X-y Zhou, M Intwala, A Kapadia, X Wang, in *NDSS*. Soundcomber: a stealthy and context-aware sound trojan for smartphones, vol. 11, (2011), pp. 17–33
48. X Zhou, S Demetriou, D He, M Naveed, X Pan, X Wang, CA Gunter, K Nahrstedt, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. Identity, location, disease and more: inferring your secrets from android public resources (ACM, New York, 2013), pp. 1017–1028
49. C Shen, H Choi, S Chakraborty, M Srivastava, in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Towards a rich sensing stack for IoT devices. (IEEE, 2014), pp. 424–427
50. L Yang, K Ting, MB Srivastava, in *Pervasive Computing and Communications (PerCom), 2014 IEEE International Conference On*. Inferring occupancy from opportunistically available sensor data (IEEE, 2014), pp. 60–68
51. S Dey, N Roy, W Xu, RR Choudhury, S Nelakuditi, in *NDSS*. Accelprint: imperfections of accelerometers make smartphones trackable, (2014)
52. E Miluzzo, A Varshavsky, S Balakrishnan, RR Choudhury, in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. Tapprints: your finger taps have fingerprints (ACM, New York, 2012), pp. 323–336
53. J Han, E Owusu, LT Nguyen, A Perrig, J Zhang, in *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference On*. Accomplice: location inference using accelerometers on smartphones (IEEE, 2012), pp. 1–9
54. S Nawaz, C Mascolo, in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. Mining users' significant driving routes with low-power sensors. (ACM, New York, 2014), pp. 236–250
55. T Watanabe, M Akiyama, T Mori, in *Proceedings of the 9th USENIX Conference on Offensive Technologies*. Routedetector: sensor-based positioning system that exploits spatio-temporal regularity of human mobilty. WOOT'15. (USENIX Association, Berkeley, 2015), pp. 6–6. http://dl.acm.org/citation.cfm?id=2831211.2831217
56. B-J Ho, P Martin, P Swaminathan, M Srivastava, in *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. From pressure to path: barometer-based vehicle tracking. (ACM, New York, 2015), pp. 65–74
57. E Owusu, J Han, S Das, A Perrig, J Zhang, in *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*. Accessory: password inference using accelerometers on smartphones (ACM, New York, 2012), p. 9
58. M Hussain, A Al-Haiqi, A Zaidan, B Zaidan, MM Kiah, NB Anuar, M Abdulnabi, The rise of keyloggers on smartphones: a survey and insight into motion-based tap inference attacks. Pervasive Mob. Comput. **25**, 1–25 (2016)
59. A Raij, A Ghosh, S Kumar, M Srivastava, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Privacy risks emerging from the adoption of innocuous wearable sensors in the mobile environment (ACM, New York, 2011), pp. 11–20
60. Y Song, M Kukreti, R Rawat, U Hengartner, Two novel defenses against motion-based keystroke inference attacks. arXiv preprint arXiv:1410.7746 (2014)