senDroid: Auditing Sensor Access in Android System-Wide

Weili Han, *Member, IEEE*, Chang Cao, Hao Chen, Dong Li, Zheran Fang, Wenyuan Xu, *Member, IEEE*, and X. Sean Wang, *Senior Member, IEEE*

Abstract—Sensors are widely used in modern mobile devices (e.g., smartphones, watches) and may gather abundant information from environments as well as about users, e.g., photos, sounds and locations. The rich set of sensor data enables various applications (e.g., health monitoring) and personalized apps as well. However, the powerful sensing abilities provide opportunities for attackers to steal both personal sensitive data and commercial secrets like never before. Unfortunately, the current design of smart devices only provides a coarse access control on sensors and does not have the capability to audit sensing. We argue that knowing how often the sensors are accessed and how much sensor data are collected is the first-line defense against sensor data breach. Such an ability is yet to be designed. In this paper, we propose a framework that allows users to acquire sensor data usages. In particular, we leverage a hook-based track method to track sensor accesses. Thus, with no need to change the source codes of the Android system and applications, we can intercept sensing operations to graphic sensors, audio sensors, location sensors, and standard sensors, and audit them from four aspects: flow audit, frequency audit, duration audit and invoker audit. Then, we implement a prototype, referred to as *senDroid*, which visually shows the quantitative usages of these sensors in real time at a performance overhead of [0.04–8.05] percent. *senDroid* allows Android users to audit the applications even when they bypass the Android framework via JNI invocations or when the malicious codes are dynamically loaded from the server side. Our empirical study on 1,489 popular apps in three well-known Android app markets shows that 26.32 percent apps access sensors when the apps are launched, and 11.01 percent apps access sensors while the apps run in the background. Furthermore, we analyze the relevance between sensor usage patterns and third-party libraries, and reverse-engineering on suspicious third-party libraries shows that 77.27 percent apps access sensors via third-party libraries. Our results call attentions to address the users' privacy concerns caused by sensor access.

Index Terms—Android security, sensor, audit, hooking, senDroid

1 Introduction

With the development of sensing technologies, smartphones are increasingly equipped with sensors, such as cameras, microphones, GPS, motion sensors, etc. [1]. Already, a modern smartphone has more than ten high-accuracy sensors, and these sensors enable novel and exciting applications.

With the trend of environment-aware and user-oriented apps, mobile applications tend to gather an increasing amount of sensor data, and the users are facing a higher risk than never before. At the beginning of 2016, it is reported that *Alipay* silently took photos without informing its users [2], although *Alipay* removed this function after users become aware of it and showed their serious concerns. Furthermore, many sensor data allow applications to infer security-sensitive information, e.g., inferring keystrokes [3] and voice [4] from motion sensors.

- *W. Han, C. Cao, D. Li, Z. Fang, and X. S. Wang are with the Software School, Fudan University and the Shanghai Key Laboratory of Data Science, Shanghai 201203, China. E-mail: {wlhan, 16212010001, 14212010008, 13212010002, xywangCS}@fudan.edu.cn.*
- *H. Chen is with the Department of Computer Science University of California, Davis, CA 95616-5270. E-mail: chen@ucdavis.edu.*
- *W. Xu is with the Department of Electronic Engineering, Zhejiang University, Hangzhou, Zhejiang 310000, China. E-mail: xuwenyuan@zju.edu.cn.*

Manuscript received 6 Nov. 2016; revised 12 Oct. 2017; accepted 13 Oct. 2017. Date of publication 1 Nov. 2017; date of current version 18 Mar. 2020. (Corresponding author: Weili Han.)
Digital Object Identifier no. 10.1109/TDSC.2017.2768536

Smartphones can audit the network traffic, yet they are incapable of measuring sensor accesses for identifying malicious or suspicious usages of sensors. An application could maliciously gather sensor data without users' consents, especially from the standard sensors (e.g., motion sensors), because they are not under the control of the Android permission mechanism. To investigate the legitimacy of an application's access to sensor data, it is necessary to monitor sensor accesses and raise an alert to the users. If inappropriate usage is found, for instance, sensor access monitoring can help to detect whether an application accesses a sensor when a user is not expecting it (e.g., secretly taking a photo), or whether it gathers an abnormally large amount of sensor data.

To the best of our knowledge, we are not aware of any framework that can comprehensively monitor the sensor data access pattern of applications, e.g., measuring when an application accesses sensors and how many data it has acquired. Although `taintDroid` [5] can detect which application is accessing a sensor, it does not measure how many data the application is reading. Measuring the sensor data access traffic is challenging, because it not only needs to be efficient and imposes small overhead but also is able to cope with circumvention by applications.

To audit the sensor access efficiently and effectively, we propose a framework, referred to as *senDroid*, to quantitatively measure sensor usages in Android. *senDroid* uses low-level hooks to intercept all sensor-related API calls so that it measures when an application reads a sensor and how many data it reads. In summary, our contributions are listed as follows:

- We design a framework, referred to as *senDroid*, which can audit all four categories of sensors: graphics, audio, location, and standard sensors, in the Android platform. *senDroid* can intercept all accesses to sensors, including access via JNI (Java Native Interface). Our performance evaluation shows that *senDroid* incurs a moderate overhead of [0.04–8.05] percent on Android smartphones. Moreover, because *senDroid* hooks into the processes of both applications and Android system to intercept sensor-related API calls, it requires neither modification to the original Android systems nor the source code of the applications. Thus, *senDroid* can be widely deployed and is capable of detecting potential attacks that bypass the Android framework.
- Our experiments show that *senDroid* can report all sensor accesses by any applications, even by the dynamically loaded codes. We evaluate *senDroid* over 1,489 popular applications in three well-known Android application markets (two popular Chinese App markets and *Google Play Store*) and study *senDroid* in two running phases of applications—*Launch Phase* and *Silent Phase*(i.e., in the background). We observe the following:
  – During the *Launch Phase*, 25.35, 11.92, 0.46, and 0.46 percent applications in Chinese Android app markets access location sensors, standard sensors, graphic sensors, and audio sensors, respectively. These numbers are 12.70, 4.24, 0.46, and 0.46 percent higher than those in *Google Play Store*, respectively. When running in *Silent Phase*, 13.19 and 2.66 percent applications in Chinese Android app markets access location sensors and standard sensors, respectively, which is 8.55 and 1.54 percent higher than those in *Google Play Store*. These results show that applications in Chinese Android app markets access sensors more than those in *Google Play Store*, especially during the *Silent Phase*.
  – We find that third-party library is one of the main causes that leads to sensor access in *Silent Phase*. We reverse-engineer the applications that access sensors during *Silent Phase*, and find that 77.27 percent applications access sensors from third-party libraries, but such accesses caused by third-party libraries rarely appear in the apps' descriptions. We recommend that developers should disclose third-party libraries and their sensor access in the app's description.

The rest of paper is organized as follows. Section 2 introduces the background knowledge and the motivated scenarios of our paper; Section 3 presents the design of *senDroid*; Section 4 shows the details of *senDroid* implementation; Section 5 evaluates *senDroid* from accuracy and performance overheads; Section 6 empirically studies the popular applications in *Wandoujia*, *360* and *Google Play Store* to show the usages of sensors in reality; Section 7 discusses the potential issues; Section 8 investigates the related research works; Section 9 summaries the paper and introduces our future work.

2 Background and Motivation

2.1 Sensors in the Android Platform

Each Android enabled devices, including smartphones, may contain more than ten sensors, such as cameras, microphones. The sensors in an Android-enabled device could be divided into four categories.

- *Graphic Sensors*. Graphic sensors refer to those sensors that collect graphic data, e.g., photos, videos. Android applications connect with graphic sensors via `CameraService` and achieve related operations, e.g., starting preview, taking a picture, or recording videos via `Camera` and `MediaRecorder`. In the Android permission system, a developer can request the permission of `CAMERA` to legally access graphic sensors.
- *Audio Sensors*. Audio sensors, e.g., microphone in the Android platform help devices to capture ambient sounds. In Android, there exists two ways to access the audio sensors. One is `AudioRecord` that provides raw sound streams to applications; and the other is `MediaRecorder` that offers compressed audio files. In the Android permission system, a developer can request the permission of `RECORD_AUDIO` to legally access audio sensors.
- Location Sensors. Location sensors primarily refer to the GPS, which collects the location data of a smartphone. Android applications utilize `LocationManager` to request the latest location or to get location periodically by registering `LocationListener`. In the Android permission system, a developer can request the permission of `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` to legally access location sensors.
- *Standard Sensors*. The Android platform officially defines the standard sensors. Some(e.g., accelerometer and gyroscope) can monitor the motion of phones' users. Others can monitor various environmental properties, e.g., humidity, illuminance, pressure, temperature, and geomagnetic field. `SensorService` is the primary service of standard sensors. Android applications obtain the list of standard sensors that the platform supports, create connection (`SensorEventConnection`) and poll the standard sensors devices through `SensorService`.

Because the sensor data usually contains rich information, the vendors of mobile applications tend to gather more and more sensor data than never before. Many literatures [6], [7], [8], [9] focus on detecting the user location via standard sensors' data instead of GPS, and on revealing sound information with standard sensors data [4], [10]. Moreover, the standard sensors' data can be utilized for inferring user's inputs [3], [11], [12], and for identifing a device [13], [14] or even an individual [15].

Android permission system utilizes permissions to control accesses to sensors. However, many standard sensors (e.g., accelerometer and gyroscope), are not under control.

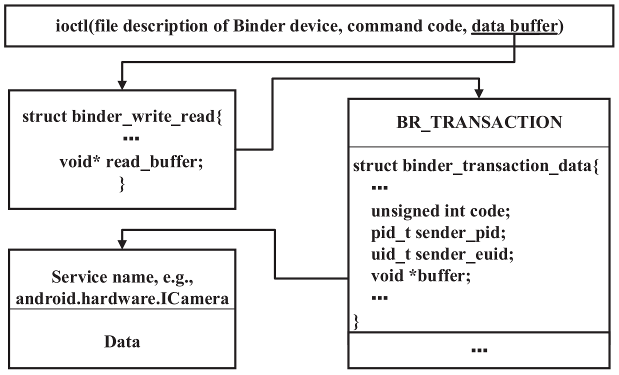

ioctl(file description of Binder device, command code, data buffer)
struct binder_write_read{
...
void* read_buffer;
}
BR_TRANSACTION
struct binder_transaction_data{
...
unsigned int code;
pid_t sender_pid;
uid_t sender_euid;
void *buffer;
...
}
...
Service name, e.g., android.hardware.ICamera
Data

Fig. 1. Brief structures in `ioctl` called by Binder.

To the best of our knowledge, it is yet to carry out the empirical study to effectively evaluate the sensor usages of mainstream application markets.

2.2 Communication Between Applications and Sensors

2.2.1 Binder

Binder [16], originally named OpenBinder, is a system-level architecture for IPC (Inter-Process Communication). Binder inter-process communication framework follows a client-server architecture, and consists of four components: client, server, service manager, and Binder driver. Binder driver communicates with a Binder device by using the system call `ioctl`. Fig. 1 indicates the related data structures and API's arguments in `ioctl`, which consists of three parts: the file description of a Binder device, a Binder driver command code, and a data buffer. The major driver command code is `BINDER_WRITE_READ`. A Binder driver uses this command to read data from or write data to a Binder device. The data read or to be written is organized in a structure named `binder_write_read`, which comprises a series of pairs of a target command and an argument. The target command we concerned is `BR_TRANSACTION` which is shown in Fig. 1. This command is sent by the client, and its corresponding argument is a request whose structure is named `binder_transaction_data`. The variable `code` in the structure of `binder_transaction_data` is the command code negotiated by the sender and receiver. In general, the serial number of public interface is defined in the service. The `uid` of an Android application that initiates the transaction is shown in the variable `sender_euid`, which can help us to identify which Android application requests the sensor data. The extra arguments, including the name of service which will deal with the request, are stored in the field `buffer`.

2.2.2 Interaction with Sensor Devices

As described in Section 2.1, there can be more than ten sensors in an Android enable device. After an application sends a request to a corresponding service, the service will interact with the corresponding sensor by calling the standard interface defined in the HAL (Hardware Abstraction Layer), which provides a standard interface for hardware vendors to implement [17]. Each accessory, e.g., sensor, in Android is treated as a file so that it can be accessed using standard I/O system calls. The sensor driver that implements the HAL interface `open`s the sensor devices and invokes `read`, `write` or `ioctl` system call to interact with the sensor devices. Android utilizes V4L2 (Video for Linux 2) as its camera driver, ALSA (Advanced Linux Sound Architecture) as its audio driver while no official GPS driver or standard sensors driver.

2.3 Hooking

Hooking is a technique of inserting codes into a system call for alteration. The typical hook works by replacing the function pointer to the call(s) a developer wants to intercept with that. Once it is done, it will then call the original function pointer [18]. There are two steps, function substitution and dynamic-link library injection, when we want to hook a system call.

2.3.1 Function Substitution

In Android, share libraries are ELF (Executable and Linkable Format) [19] files which are mapped into the memory space of a process at runtime. The GOT (Global Offset Table) and the PLT (Procedure Linkage Table) are two pivotal parts of ELF file. A GOT is a table of addresses in the data section. When an instruction in the code section refers a variable, it looks up the entries in the GOT, which keeps the absolute addresses of variables. Each entry of the PLT is a chunk of instructions corresponding to an external function the shared library calls [20]. When an instruction calls an external function, it calls an entry in the PLT, which then calls the actual function. The address of the actual function is determined by the corresponding entry in the GOT.[1] The instructions to call external function are essentially jump instructions pointing to entries in the PLT. Then, the PLT entries retrieve the absolute addresses of the functions, which are contained by the GOT entries. Due to the indirection to function references, the hooking can be achieved by substituting designate function pointers with the original ones in the GOT for each ELF file that the target process loaded.

2.3.2 Dynamic-Link Library Injection

Dynamic-link library (DLL) injection is a technique that allows a process to run codes in the address space of another process by forcing it to load a dynamic-link library [21]. In Android, an approach to realize DLL injection is to leverage the system call `ptrace`, which provides a process with the capability of monitoring and controlling the execution of another process [22].

2.4 Motivation Scenarios

`Alice` bought an Android phone with several applications pre-installed. She might also want to install several applications herself. The recent news reported that applications could steal information from on-board sensors in mobile devices. Worried that the applications on her phone could be malicious, she installed our *senDroid* on her phone. After running *senDroid* for several days, she can check the reports

1. The address contained by the corresponding GOT entry points to the PLT entry itself when first calling the function. It points to the actual function only when the dynamic loader resolves it. This mechanism is called lazy binding or lazy linking [20] while it is not adopted by the current version of Android.

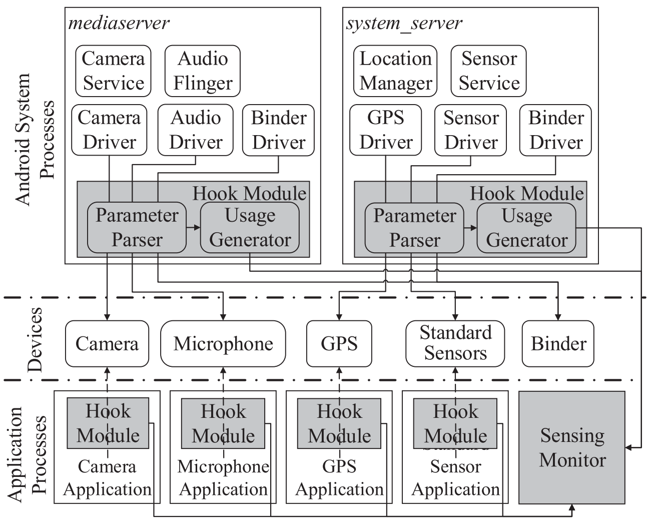


Fig. 2. Design of *senDroid* framework. The shadowed blocks are designed in *senDroid*.

of *senDroid*, and judge whether the installed applications stole sensors data. *senDroid* can help users to monitor and analyze the usages of sensors on a mobile device, with only negligible performance overhead but without the modification of Android framework.

In another scenario, `Bob`, who is a bouncer of an application market of Android, can use *senDroid* to detect suspicious applications in the market. After installing *senDroid*, he could install the suspicious applications on an Android phone, and operate the applications with the help of test tools. Then he may compare the log with the applications' description either manually or automatically with the supports of natural language processing.

3 senDroid: Design

3.1 Threat Model

The security threats considered in our work come from the third-party applications which access sensors data surreptitiously. These applications can be divided into two types:

- Malwares, e.g., applications implementing the potential attacks proposed in prior work [4], [23], [24], [25]. Some of them infer personal privacy from the unrestrained sensors(e.g., accelerometer and gyroscope), while the others access restrained sensors by an undetectable approach, e.g., taking photos or recording video by JNI without calling any Android API. Even some applications dynamically load remote codes to implement suspicious functions bypass applications' bouncers.
- Applications [26] using ad/analytics libraries that collect sensor data for precision advertising and improving user experience. But they do not declare their accesses in their description.

Note that, both types of applications can be suspicious but not vicious, which means that they would not require a highly-escalated privilege or circumvent the Android system(e.g., modifying or bypassing the SELinux policies on the device). Targeted at these honest but curious applications, and motivated to mitigate the above threats, we propose *senDroid* to counter these suspicious behaviors by auditing the accesses to sensors on an Android device.

TABLE 1
Audit capacities of *senDroid* for Four Categories Sensors

Sensor Type	Sensor Data	Flow Audit	Frequency Audit	Duration Audit	Invoker Audit
Graphic Sensors	Preview	√	√	√	√
	Video	√	√	√	√
	Photo	-	√	-	√
Audio Sensors	Audio	√	√	√	√
Location Sensors	Location	√	√	√	√
Standard Sensors	Motion and Environment	-	√	√	√

3.2 Framework Overview

Fig. 2 shows an overview of sensing audit in *senDroid*. Basically, *senDroid* consists of (1) *Hook Module*, a sentry deployed in system or application processes that intercepts the sensor-related traffic to reveal sensing usage; and (2) *Sensing Monitor*, an Android application that visualizes the sensing usages. The solid arrows in Fig. 2 represent the standard access mode of sensors. In the standard access mode, sensors applications send requests to corresponding services, and the services handle the sensors' data accesses. The dashed arrows represent a deviant access mode in which sensors applications directly access the sensor device or call the interface defined in HAL in virtue of JNI. To handle both standard and deviant access modes, the *Hook Module* is embedded in both Android system processes that contain sensors' service and application processes that request sensor data. Overall, *senDroid* interposes in the sensor data flow to perform four types of audit.

- *Flow Audit*. Every time accessing a sensor, an application gains the sensor data in various size. We monitor the size of sensor data to report the amount of sensor traffics the application requested the sensor.
- *Frequency Audit*. When accessing sensor data, an application may send requests with different frequencies. *senDroid* can reveal how often and what time the application accessed the sensor data.
- Duration Audit. In the continuous sensing case, we record the duration to profile the behavior of applications more comprehensively.
- *Invoker Audit*. *senDroid* audits which application initiates the sensor data access originally.

The audit capacities of *senDroid* for different categories of sensors are shown in Table 1. We will discuss the reasons why *senDroid* does not implement the flow audit of photo and standard sensors in Section 4.

3.3 Hook Module

A *Hook Module* consists of two main components: a *Parameter Parser* parses the data in intercepted system calls and a *Usage Generator* generates the usage reports of sensors according to the accessing time, data size and sensor accessor. Table 2 illustrates APIs that *Hook Module* intercepts, which parameters of these APIs that *Hook Module* parses, and which audit types the parsed parameters are leveraged to perform. The details of hooking are described in Section 4.

TABLE 2
APIs and Parameters Related to the Sensing Analysis in *senDroid*

Sensors Type		Flow Audit	Frequency Audit	Duration Audit	Invoker Audit
Graphic Sensors	Preview	`ioctl:DQBUF`		`ioctl:STREAMON/OFF`	`ioctl:BINDER_WRITE_READ:` Service name and `sender_euid`
	Video			`ioctl:BINDER_WRITE_READ:ICamera:START/STOP_RECORDING/IMediaRecorder:START/STOP`	
	Photo	-	`ioctl:BINDER_WRITE_READ:ICamera:TAKE_PICTURE`	-	
Audio Sensors		`ioctl:READI_FRAMES`		`ioctl:BINDER_WRITE_READ:IAudioRecord/IMediaRecorder:START/STOP`	
Location Sensors		`msg_q_rcv:REPORT_POSITION`		`ioctl:BINDER_WRITE_READ:ILocationManager:REQUEST/REMOVE_UPDATES`	
Standard Sensors		-	`ioctl:BINDER_WRITE_READ:SensorEventConnection:SET_EVENT_RATE`	`ioctl:BINDER_WRITE_READ:SensorEventConnection:ENABLE/DISABLE`	

Parameter Parser. The submodule of *Hook Module* is a set of functions that we implement to substitute the system calls that are related to sensor accessing. These functions intercept all the communication requests sent from an application to sensor devices or Binder. When a system call is intercepted, *Parameter Parser* parses the parameters carried by the system call. According to the parsing result, the system calls are delivered to *Usage Generator* for further analysis.

Usage Generator. Leverages a set of audit policies to generate the sensing usage reports. Each entry to report sensing usage is defined as a vector, which includes the accessing time, data size and sensor accessor. The information delivered from *Parameter Parser* is sporadic. So we need to gather the information to create sensing usage reports, which are stored in a local database. These reports are available to be accessed by *Sensing Monitor*.

3.4 Sensing Monitor

Sensing Monitor is an application that allows a user to analyze the sensing usages generated by *Hook Module*. *Sensing Monitor* can visualize the reports of the sensing usages by applications or by sensors in a comprehensive manner. Fig. 3 shows one user interface of *Sensing Monitor*. A graph of location sensors usage is presented to the user, where the red curve denotes the overall usage over the time. The usage of each application is represented by curves in different colors.

4 SENDROID: IMPLEMENTATION

We leverage the graphic sensors as an example to illustrate the implementation of *senDroid*:

Fig. 4 shows the interposition of *senDroid* in the data flow of the camera's APIs, e.g., `startPreview`. The application process does not actually communicate with the camera device. Instead, the interaction with the camera device is implemented in the *mediaserver* process. The application process informs the *mediaserver* process of its request through Binder. When receiving the request sent by the application process, the camera service and camera driver pass the request to the camera device. The camera service also returns the execution results via Binder.

The interaction between the *mediaserver* process and Binder indicates that the request is sent by which application and is delivered to which service. The interaction

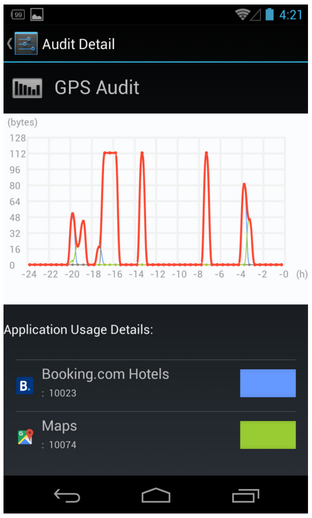


Fig. 3. Visual demonstration of sensing usages. The report includes the total usages for all applications, and the individual usage of each application.

between the *mediaserver* process and the camera device indicates what and how many data the application obtains. These two interactions are the choke points of the data flow and are where *senDroid* interposes.

4.1 Interception Between Service and Binder

We leverage *senDroid* to intercept interactions between the service and Binder. Concretely, the *Hook Module* hooks the `ioctl` called by Binder. Then, the *Parameter Parser* parses the `read_buffer` in the structure `binder_write_read`, and finds out all the `BR_TRANSACTION` commands(illustrated in Section 2.2.1). Furthermore, we extract the receiver and the requester from the first entry of `buffer` in the structure `binder_transaction_data` and the field `sender_euid`, respectively. Finally, we identify the operation according to the field `code`.

4.2 Interception Between Service and Devices

4.2.1 Graphic Sensors

senDroid intercepts the `ioctl` calls with the commands `VIDIOC_STREAMON` and `VIDIOC_STREAMOFF` to learn when an application opens and closes cameras. When starting preview, the camera driver calls `ioctl` with the command `VIDIOC_DQBUF`, and the corresponding data buffer

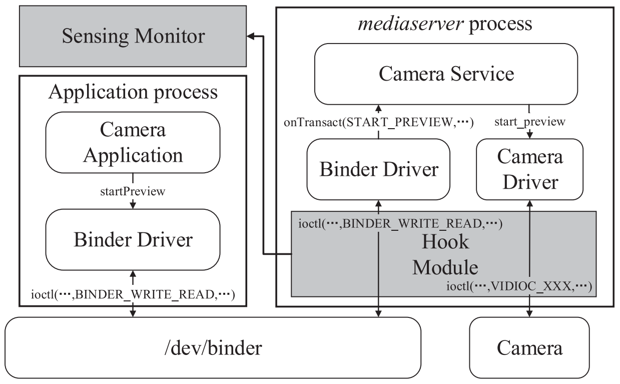


Fig. 4. Interposition of *senDroid* in the data flow of `startPreview`. The shadowed blocks are designed in *senDroid*.

is a structure named v4l2_buffer. This structure contains variables, from which we can infer the size of a graphic frame. At the driver layer, the preview, the photo, and the video are all graphic frames so that we cannot pick out a photo or a piece of video from the stream of graphic frames. Fortunately, we can demarcate the stream according to the Binder requests sent by the applications. The ioctl calls called by Binder sending request to ICamera with the code TAKE_PICTURE indicates that an application is taking photos. Similarly, Binder request sent to ICamera or IMediaRecorder with the code START/STOP_RECORDING extracts a piece of video from the stream of frames. It is noteworthy that an application can take pictures or record videos using PreviewCallback instead of takePicture or MediaRecorder. *senDroid* monitors Binder request sent to ICamera with the code SET_PREVIEW_CALLBACK_FLAG to learn whether a PreviewCallback is registered before the preview starts or during previewing.

4.2.2 Audio Sensors

senDroid intercepts the ioctl calls invoked by the audio driver with the command SNDRV_PCM_IOCTL_READI_FRAMES. We calculate the size of audio record from the corresponding data buffer named snd_xferi. ALSA only has the command SNDRV_PCM_IOCTL_START, with no command SNDRV_PCM_IOCTL_STOP. So we obtain the start and end time of the audio record by the approach described in Section 4.1. Concretely, we capture the ioctl calls called by Binder where the service name is android.media.IAudioRecord or android.media.IMediaRecorder with the code either START or STOP. We use the time of these calls as the start and end time of the audio record.

Considering that the MediaRecorder can be utilized to record audio as well as video, and before using MediaRecorder to record audio or video, applications are required to invoke setAudioSource and setVideoSource respectively, we distinguish these two usages by monitoring the invocation of setAudioSource and setVideoSource. The corresponding codes in the ioctl called by Binder are SET_AUDIO_SOURCE and SET_VIDEO_SOURCE respectively.

4.2.3 Location Sensors

When the location information is sent via the message queue mechanism, which is illustrated in Section 2.2.2, *senDroid* hooks the message receiving calls called by the GPS driver and parses the message received from the GPS device. We determine the operation specified by the message according to the message id. *senDroid* intercepts the messages with the message id REPORT_POSITION, which indicates that the received data are geographic positions reported to a location requester. We intercept the ioctl calls where the service name is android.location.ILocationManager to determine the start and end time of the tracking of GPS according to the code REQUEST_LOCATION_UPDATES and REMOVE_UPDATES.

4.2.4 Standard Sensors

As mentioned in Section 2.2.2, standard sensors have no open source driver on Nexus 4. So we cannot learn how standard sensors driver communicates with corresponding device. However, *senDroid* can intercept the ioctl calls called by Binder that sending request to SensorEventConnection with the code ENABLE_DISABLE. This Binder request carries two parameters: a standard sensor handle and a Boolean (TRUE presents enabling and FALSE presents disenabling). So we can know which standard sensor is activated and when it is activated or deactivated. Moreover, Binder request with code SET_EVENT_RATE contains a parameter that indicates the access frequency of standard sensors application. Because the data size of one standard sensor record is negligible. So although we cannot obtain the exact data of standard sensors, we argue that activated time interval and access frequency of standard sensors are crucial and sufficient to audit the sensing based on the standard sensors.

4.3 Prototype Setup

We use the Nexus 4 with Android 4.2.2 as our experiment platform. When we set up our prototype, we need to root the device first, and then leverage the existing DLL injection approach[2] to inject into the target processes (*mediaserver*, *system_server* and application processes that access sensors[3]), load the function substitution codes into the target processes and execute the function substitution. Then, we leverage the implementation in [27] to accomplish the function substitution. The function substitution code goes through the memory map of the target processes and load each ELF file to substitute the functions we concerned, e.g., ioctl for our own functions. After deploying *senDroid*, all four categories of sensors, including camera, microphone, GPS, and standard sensors, will be audited. The sensing usage is logged in XML format for subsequent analysis in *Sensing Monitor*.

5 senDroid: Evaluation

In order to evaluate the accuracy and overhead of *senDroid*, we set up the application dataset which consists of 540 applications downloaded from the top free chart of *Google Play Store* in April, 2016. We only keep the applications which require access to sensors monitored by *senDroid* while remove the others. Of the 540 applications, 429 applications require access to one or more of the four categories of sensors. Further, We ignore applications which have internal failure or whose sensor related functionalities cannot be reached due to geological or system version restriction. For camera, microphone and GPS, we determine whether an application requires access to these three categories of sensor information based on the permissions it requests. For standard sensors, we determine that an application requires access to standard sensor data if it declares <uses-feature> tag for android.hardware.sensor.*.

We randomly select three ones for each type of sensors under the auditing of *senDroid*. All evaluation experiments

2. https://github.com/shutup/libinject2
3. The camera service and audio service are launched in *mediaserver*, the location service and sensors service are launched in *system_server*. We cannot guarantee whether an application directly accesses sensors via JNI, so we inject into every application process.

TABLE 3
Comparison Between the Time of Accessing Sensors Observed Manually and the Time of Accessing Sensors Recorded by *senDroid*

Action	App	Mean start time $\|\Delta\|$ (s)	Mean stop time $\|\Delta\|$ (s)
Taking pictures	PicsArt	0.00	-
	Poshmark	2.00	-
	Perfect365	0.66	-
Taking video	Zoosk	1.00	0.50
	Tango	0.50	0.50
	ooVoo	1.00	0.50
Recording audio	Tom Loves Angela	0.00	0.00
	Tango	0.00	0.00
	Talking Tom	0.00	0.00
Requesting location	CM Security	0.50	0.00
	Expedia	0.00	0.00
	360 Security	0.00	0.00
Reading standard sensors	Temple Run 2	4.00	2.50
	Bowling Kings	2.00	2.50
	Traffic Racer	3.00	3.00

are run on a Nexus 4 with 2 GB RAM running Android 4.2.2.[4]

5.1 Deviation of Data Reported by *senDroid*

5.1.1 Experiment Results

Table 3 shows the experiment results for applications which use camera to take pictures or take videos, use microphone to record audio, use GPS or network to request locations, and access standard sensors. We record the start time and end time of each *Action* on each *App*. We repeat the above operation three times. Then, in Table 3, *Mean start time* $|\Delta|$ refers to the average value of start time difference between the value recorded by a tester and by *senDroid*. And *Mean stop time* $|\Delta|$ refers to the average value of end time difference between the value recorded by a tester and by *senDroid*.

Applications listed in Table 3 respectively provide users with the functions of taking pictures, taking videos, recording audios, requesting locations, and reading standard sensors. We record the time when we trigger the actions and the time when *senDroid* detects that the application executes the actions. Considering the time from UI operations to the time the application actually executes the actions, the deviation is acceptable. As is shown Table 3, *senDroid*'s deviation for detecting access to standard sensor data is larger than that of other sensors. Because games will usually play animations before and after the game starts or ends, so it is more difficult to infer the time when applications start or stop requesting standard sensor data.

4. Although the new version of Android has many new features, the technical details of sensor access are almost same. We, thus, use this version as our experiment platform. Furthermore, these popular applications may run this version of Android, and be empirical studied according to sensor usages in the next section.

TABLE 4
Comparison Between the Actual Length of Videos and Audios and the Length Which *senDroid* Records

	Lengths(s)			
	Actual	Average recorded by senDroid(Differences)		
		Device 1	Device 2	Device 3
video	10.00	11.30(1.30)	11.31(1.31)	11.34(1.34)
	30.00	31.32(1.32)	31.28(1.28)	31.31(1.31)
	60.00	61.31(1.31)	61.38(1.38)	61.33(1.33)
	120.00	121.24(1.24)	121.31(1.31)	121.31(1.31)
audio	10.00	10.21(0.21)	10.21(0.21)	10.20(0.20)
	30.00	30.19(0.19)	30.19(0.19)	30.21(0.21)
	60.00	60.21(0.21)	60.25(0.25)	60.23(0.23)
	120.00	120.23(0.23)	120.24(0.24)	120.23(0.23)

5.1.2 Accuracy of Recorded Video & Audio Length

We developed an evaluation application to measure the length of videos and audios which *senDroid* records, and compared them with the actual lengths. We leveraged the `setMaxDuration` API of the class `android.media.MediaRecorder` to set the length of the videos and audios. After the recording reaches the length we set, `MediaRecord` will stop the recording. Here, we deploy *senDroid* on three different Nexus 4 in order to explore whether the deviation differs on different devices. For each value of length, we ran the test for five times and calculated the average length recorded by *senDroid*.

As is shown in Table 4, the lengths which *senDroid* record are about 1.3 seconds and 0.2 second more than the actual lengths of the video and audio, respectively, and the deviation is almost the same on different devices. Moreover, as the lengths of the video and audio increase, the deviation almost remains unchanged. According to our investigation, the deviation may be caused by the overhead of inter-process communication or the preprocessing and postprocessing of the record. Since the deviation is consistent for the same type of sensor across different devices and different lengths of recordings, we ignore the deviation in the length of recordings reported by *senDroid*.

5.2 Audit of Taking Pictures from Preview Frames

As we described in Section 4.2.1, an application can process the provided preview data in the `onPreviewFrame` callback. This means that applications can take pictures or record videos without calling the `takePicture` or `MediaRecorder` API by starting camera preview and taking screenshot programmatically.

To better evaluate the effectiveness of *senDroid* with this case, we evaluated *senDroid* with an application called Spy Camera HD.[5] According to Spy Camera HD's description on *Google Play Store*, it allows users to secretly take photos without any shutter sound and camera preview on the phone screen. Users can instruct Spy Camera HD to take pictures by shaking the phone, whistling or setting a timer. After reverse-engineering Spy Camera HD, we confirm that Spy Camera HD actually takes pictures in the

5. https://play.google.com/store/apps/details?id=com.usefullapps.spycamera

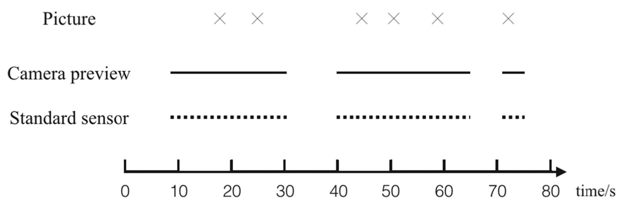


Fig. 5. The timestamps of pictures taken by Spy Camera HD and the time durations of Spy Camera HD's usage of sensors detected by *senDroid*.

`onPreviewFrame` callback it registers by processing the content data of the preview frame which is provided as an argument when the `onPreviewFrame` callback is called. Even on devices which shutter sound is enabled forcibly, capturing the provided preview data in `onPreviewFrame` instead of calling `takePicture` directly will not trigger any shutter sound. In this way, Spy Camera HD can take pictures without people's awareness.

We used Spy Camera HD to take several pictures by shaking the phone and matched the time when the pictures were taken based on the pictures' timestamps with the duration of camera preview and the duration of application requesting standard sensor data which *senDroid* records. The result is shown in Fig. 5. The first row shows the time which Spy Camera HD took a picture based on the timestamp of the saved picture file. For example, the first picture was taken at the 18th second. The second and third rows show the time duration when Spy Camera HD started the camera preview and requested standard sensor data respectively. For example, the first duration of camera preview started at the 9th second and ended at the 30th second, which also matches the first duration of Spy Camera HD's requesting for standard sensor data. During the test, we took five pictures, and there were three durations when Spy Camera HD started camera preview and requested standard sensor data at the same time. We can conclude from Fig. 5 that all pictures taken with Spy Camera HD fall in the camera preview duration and standard sensor data request duration which *senDroid* records.

Since *senDroid* will not only record applications' calling the `takePicture` API, it will also detect applications which override the `onPreviewFrame` callback and record the duration which applications start the camera preview, these two tricky methods of taking pictures can be detected by *senDroid*. With *senDroid*, users can have a complete overview of how applications access the camera and be aware of potential suspicious usage of camera and other sensors.

5.3 Audit of Dynamically Loaded APK

In February 2016, *Alipay*, backed by Chinese e-commerce giant Alibaba, was accused of taking pictures and recording audios secretly with its Android client [2]. As discovered by Twitter user typcn [28], the code logic of *Alipay* for Android's suspicious behavior consists of a file with the extension `.so` downloaded from a remote server via the Internet. The `.so` file is actually an executable APK file, which will be run as a plugin inside *Alipay* for Android. Since the `.so` file will be checked for updates periodically, the suspicious behavior was quickly removed quietly after the incident became popular on the Internet without users' updating the host *Alipay* for Android application itself.

TABLE 5
Macrobenchmark Results

Action	w/o *senDroid* (s)	w/ *senDroid* (s)	Overhead
Picture	2.38(0.09)	2.58(0.14)	8.05%
Video	115.80(0.46)	115.84(0.14)	0.04%
Audio	103.32(0.07)	103.37(0.08)	0.04%

We built an application which emulates the dynamic APK loading feature of *Alipay* for Android based on the open source project ACDD [29]. This application can download APK files as plugins from a remote server, extract DEX files from the APK files and finally run the code logic in the APK files using `DexClassLoader`, which is similar to the claimed mechanism leveraged by *Alipay* for Android. Plugins are compiled with a patched *aapt* tool, so the resource IDs of plugins will not conflict with those of the host application, and the host application can distinguish resources from different plugins by reading the first byte of the resource ID. The `mInstrumentation` variable of the `android.app.ActivityThread` class is also hooked, so that when components such as activities and services in the plugins are launched, corresponding resources are loaded. The host application which we built is a dummy application which only loads and runs APK files from a remote server, while a dynamically loaded APK file contains the logic of taking pictures and recording audios. As we tested, *senDroid* successfully detected the usage of camera and microphone in the dynamically loaded APK.

5.4 Performance Evaluation

5.4.1 Performance of Sensor Operations

To measure the performance overhead of calling the sensor APIs incurred by *senDroid*, we built a test application to take pictures, record videos, and record audios successively, and compared the time which it takes to perform the same operations with *senDroid* installed or not. For the operation of taking pictures, we takes 10 pictures successively. The measurement includes the time from executing the first taking picture action by calling the `takePicure` API to the calling of the `onPictureTaken` callback for the last picture. Note that, to better reflect the actual overhead of calling the camera APIs incurred by *senDroid*, all pictures are dropped without saving to the disk, so the time of disk I/O is eliminated. For the operation of recording video and audio, the customized application records 10 pieces of video/audio with a length of 10 seconds. This measurement includes the time from preparing recording the first video/audio to the calling of the `onInfo` callback with a "what" code of `MEDIA_RECORDER_INFO_MAX_DURATION_REACHED` for the last video/audio. Due to restriction of the `MediaRecorder` API, video and audio cannot be recorded by `MediaRecorder` without saving, so the results include the time of disk I/O. Since both the `takePicure` API and the `MediaRecorder` API are IPC-based APIs which requires *senDroid* to extract information from the `ioctl` system call, the evaluation results can truly reflect *senDroid*'s performance overhead in the worst cases.

All tests were performed on a same newly flashed Nexus 4 with 2 GB RAM running Android 4.2.2 after a reboot in the same environment. Table 5 shows the results. The numbers

TABLE 6
AnTuTu Benchmark Results (Larger Numbers in Cells Imply More Efficient Performance for Test Cases)

	Without *senDroid*	With *senDroid*	Overhead
RAM	5058.4(218.9)	5040.6(167.7)	0.35%
CPU mathematics	3600.4(58.8)	3558(25.4)	1.18%
CPU common use	3692.8(305.0)	3442.6(116.3)	6.78%
CPU multi-core performance	2511.8(67.7)	2526(30.5)	-0.57%
UX data security	1270.4(72.5)	1291.4(8.3)	-1.65%
UX data processing	549.6(12.2)	541.4(7.9)	1.49%
UX strategy games	754.8(20.1)	750.4(11.5)	0.58%
UX image process	259.4(5.7)	260.2(3.9)	-0.31%
UX I/O performance	1487(19.5)	1527.6(33.6)	-2.73%
Overall	19184.6(212.985)	18938.2(200.3)	1.28%

in parentheses indicate the expected range of values with a confidence interval of 95 percent. *senDroid* adds approximately 8.05, 0.04 and 0.04 percent overhead to taking pictures, recording videos and recording audios, respectively. The additional overhead can be attributed to storing the sensor usage to database. Thus, we can conclude that the performance overhead imposed by *senDroid* is negligible.

5.4.2 AnTuTu Benchmark

Further, we want to know whether *senDroid* will also introduce negligible overhead to the overall system or not. We use a popular Android benchmarking tool: AnTuTu [30] to compare the system performance with and without *senDroid*. The average benchmark results of five tests for a Nexus 4 running Android 4.2.2 with and without *senDroid* respectively are shown in Table 6. Similarly, the number in parentheses indicating the expected range of values with a confidence interval of 95 percent. We can conclude from Table 6 that *senDroid* also imposes negligible overhead with only 1.28 percent on the overall system performance.

6 SENDROID: EMPIRICAL STUDY

6.1 Experiment Setup

In this section, we conducted an extensive, empirical study on the sensor usage in real applications from popular markets. Before the study, we collected applications from *Wandoujia*, *360* and *Google Play Store* in September and October 2016, among which Wandoujia and 360 are predominant application markets in China. We mainly picked top free applications of each category, and the category distributions of applications in each application market are shown in Tables 7, 8 and Table 9 respectively. Note that, there exists an applications' overlap between these three markets, which means that a same application can be tested twice or three times. Especially, there are 119 applications being tested both in *Wandoujia* and *360*. Basically, we focus our study on the accesses to sensors of the applications in the following two phases:

- *Launch Phase*: Once an application is launched, it may detect the availability of the concerned sensors or gather sensors data for initialization and cause a large amount of accesses to sensors. We install and start all applications in dataset on a smartphone one by one. Here, we define a *Launch Phase* interval for 2 minutes just after each application started, and *senDroid* will audit the accesses to sensors during this interval and write the report in a log file.
- *Silent Phase*: We also evaluate the accesses to sensors when an application is running in the background. The study for *Silent Phase* can help to reveal the

TABLE 7
Application Category Composition of *Wandoujia* Dataset

Category	Number	Category	Number
COMMUNICATION	30	UTILITY	29
EDUCATION	30	GAMES	28
BEAUTY&BABY	30	TRANSPORTATION	28
MUSIC	30	TOOLS	28
LIFESTYLE	30	SHOPPING	27
PHOTOGRAPHY	30	PRODUCTIVITY	25
NEWS&MAGAZINES	30	VIDEO	23
PERSONALIZATION	29	SOCIAL	15
FINANCE	29	HEALTH&FITNESS	3
TRAVEL	29		

There are 503 Applications from 19 Categories.

TABLE 8
Application Category Composition of *360* Dataset

Category	Number	Category	Number
FINANCE	30	COMMUNICATION &SOCIAL	28
PHOTOGRAPHY	30	WALLPAPER	27
LIFESTYLE	30	BUSSINESS	27
GAMES	30	EDUCATION	22
NEWS& MAGAZINES	29	SYSTEM SECURITY	21
SHOPPING	29	HEALTH&MEDICAL	15
MAPS&TRAVEL	29	MUSIC&VIDEO	14

There are 361 Applications from 14 Categories.

TABLE 9
Application Category Composition of *Google Play Store* Dataset

Category	Number	Category	Number
TOOLS	35	NEWS_AND_ MAGAZINES	25
COMICS	34	PRODUCTIVITY	23
PERSONALIZATION	34	HEALTH_AND_ FITNESS	22
BOOKS_AND_ REFERENCE	32	MEDICAL	22
WEATHER	31	GAMES	20
BUSINESS	29	LIFESTYLE	20
SOCIAL	29	TRAVEL_AND_LOCAL	20
COMMUNICATION	27	ENTERTAINMENT	19
EDUCATION	27	LIBRARIES_AND_ DEMO	19
PHOTOGRAPHY	27	MEDIA_AND_VIDEO	19
MUSIC_AND_AUDIO	26	SPORTS	17
SHOPPING	26	FINANCE	16
TRANSPORTATION	26		

There are 625 Applications from 25 Categories.

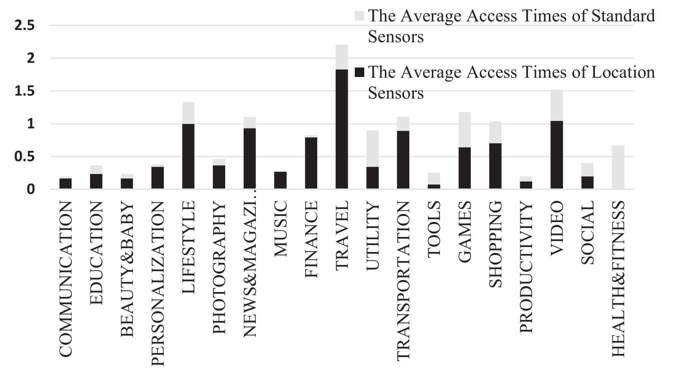


Fig. 6. The average access times during *Launch Phase* to each type of sensors in each category of *Wandoujia* market.

applications' malicious use or abuse of sensors in current prevailing application markets. In this phase, any sensor usage recorded by *senDroid* will be regarded as suspicious usage. Due to the memory limitation of the tested Android device, we run 16 applications in the background at the same time for about 24 hours, thus to make sure that we would get accurate and enough information about sensor usages.

Note that, although it should be more efficient to run evaluation tests on emulators, audio sensors and standard sensors are unfortunately disabled and cannot be emulated on emulators [31], [32]. Thus we must conduct the experiments manually on real devices.

6.2 Experiment Results of Empirical Study

We studied applications downloaded from *Wandoujia*, *360* and *Google Play Store* respectively from October, 2016 to January, 2017. Because the numbers of applications in different markets and categories are not uniform, to eliminate the interference, we calculate the percentage of applications in the results.

6.2.1 *Launch Phase*

For *Wandoujia*, we got 2,209 records of sensor accesses, among which 396 records were generated during *Launch Phase*. The accesses mainly concentrate on GPS and standard sensors: There are 117 (23.3 percent) applications under 18 different categories out of 503 valid samples access location sensors in *Launch Phase*. The accesses to location sensors are very widespread. As is shown in Fig. 6, the

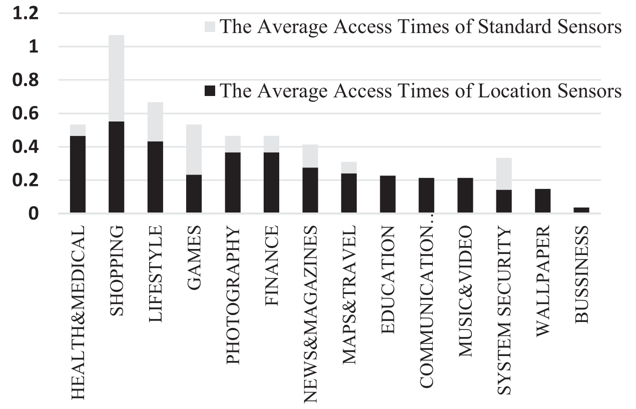


Fig. 7. The average access times during *Launch Phase* to each type of sensors in each category of *360* market.

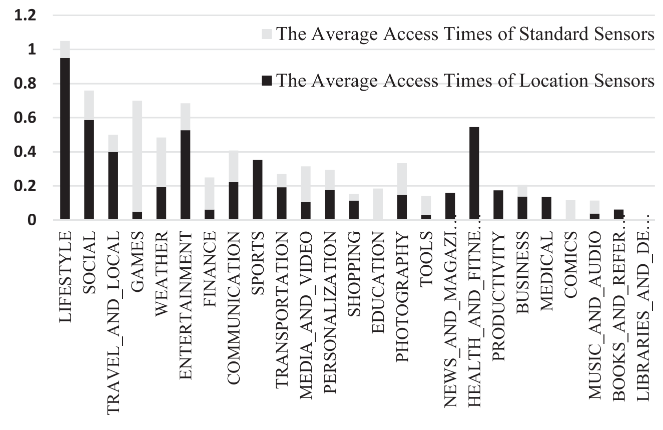


Fig. 8. The average access times during *Launch Phase* to each type of sensors in each category of *Google Play Store* market.

applications under the categories of *Lifestyle*, *Travel* and *Shopping* are most active to access location sensors in this phase. Besides,about 12.3 percent of applications try to access various kinds of standard sensors during the *Launch Phase*. Differing from location sensors, standard sensors are typically accessed by applications of *Health&Sports*, *Games* and *Videos*. We also have an inspection into the accessing frequency of each kind of standard sensors during this phase. Among all the six types of standard sensors, accelerometer is the most popular one and the number of its related records is far more than the records of other types of standard sensors. Note that, few application try to fetch the graphic sensors data or audio sensors data in this phase, and the only four applications are basically scan tools or shooting tools, which depends their main functions on the use of camera preview.

For *360*, there are 103 and 42 applications accessed location sensors and standard sensors respectively, while only one accessed graphic sensors and one accessed audio sensors. Comparing with *Wandoujia*, the ratio of location sensors usages in *360* is a little bit higher while the ratio of standard sensor usage is lower. We calculate the average access times to each type of sensors in each category respectively as shown in Fig. 7. Similarly, the applications in *Shopping* and *Lifestyle* categories still behave actively not only on accessing location sensors but also on accessing standard sensors. Over 55 percent of the applications in *Shopping* category triggered location sensors access, and 51.7 percent of *Shopping* applications accessed standard sensors.

At last, for the applications in *Google Play Store*, the overall accessing during *Launch Phase* is relatively less than applications from the above two markets. During the *Launch Phase*, only 79 out of 625 applications accessed location sensors, and 56 applications were recorded with standard sensors' accessing history. Similarly, the accesses to accelerometer are the most frequently among all types of standard sensors, while there are no access record for light and proximity sensor data. An overall distribution of categories that accessed sensors during *Launch Phase* is shown in Fig. 8.

6.2.2 *Silent Phase*

We carefully reviewed all the records and filtered out the applications which accessed sensors during the *Silent Phase* for each markets respectively, since any usage in this phase

TABLE 10
Percentages of Sensor Usages in Three Different Markets

	Overall	*Launch Phase*	*Silent Phase*
Wandoujia	33.40%	28.82%	13.92%
360	37.67%	34.90%	16.62%
GooglePlay Store	22.40%	19.36%	5.44%
Total	29.82%	26.32%	11.01%

TABLE 11
Percentage of Location Sensors and Standard Sensors in Different Markets in Different Stages

	GPS	Standard Sensors
Wandoujia	23.26% / 12.33%	12.13% / 1.99%
360	28.25% / 14.40%	11.63% / 3.60%
Google Play Store	12.64% / 4.64%	7.68% / 2.66%
Chinese App Markets	25.35% / 13.19%	11.92% / 2.66%
Overall	20.01% / 9.60%	10.14% / 2.01%

The data is presented in the format of usage percentage in Launch Phase/ usage percentage in Silent Phase.

can be regarded as suspicious usages. Still, no matter in which one of the markets, accesses to location sensors prevail and the next is standard sensors. There are 13.9 percent applications in *Wandoujia* accessing sensors during *Silent Phase* while for *360* the ratio is 16.6 percent. However, for *Google Play Store*, the ratio is far more lower. Only 5.4 percent of applications in it tried to access sensors. Interestingly, we find that quite a portion of applications access the sensors with a certain time interval and a fixed accessing period. This interval can be one hour, or two hours and the access may last for 2 seconds or even longer. We further study the reason why it happens and we would give a deeper discussion about suspicious usages in *Silent Phase* in Section 6.3.

6.2.3 Comparison and Summary

After the data collection and statistics, we analyze the results of both *Launch Phase* and *Silent Phase*, and make a comparison of the overall sensor usages in these three markets in Table 10. We can read from the table that sensors are used widely in *Wandoujia* and *360*, even in *Silent Phase*. And the usages of location sensors and standard sensors in different stages are also given in Table 11. Besides, both the usage of graphic sensors and the usage of audio sensors are less frequent, and the percentages are less than 1.0 percent, no matter in *Launch Phase* or *Silent Phase*. For *Google Play Store*, there is an observable gap between its usage ratio and the ratio of the other two markets. Considering that the applications we collected are applications on the top free charts, and the dominant user group of these three markets are different, we can reasonably conclude that the users of *Google Play Store* may be more likely to download the sensor-friendly applications.

Sensors are widely used in various categories of applications. In *Wandoujia*, over 60 percent of the applications in *Travel* and *Lifestyle* ever accessed sensors during our experiments, no matter in *Launch Phase* or *Silent Phase*. While nearly half of the applications under *Shopping*, *Video* and *Games* used sensors more or less. In *360*, *Health&Medical*, *Shopping*, *Lifestyle* and *Games* are in the front rank. At last, in *Google Play Store*, applications in *Lifestyle*, *Social* and *Travel&Local* behaved actively. Although the category lists of each application market are not totally the same, we can still find that certain categories of applications behave more actively than the others. For example, *Lifestyle* and *Games* are two representative categories in which sensors are extensively accessed.

6.3 Analysis and Case Study

As we mentioned above, any sensor access in *Silent Phase* is regarded as suspicious usage. In order to find out the applications which generated a huge amount of sensor data in our experiment, especially during *Silent Phase*, and how the sensor data would be used, we drill down into every application that accesses sensors in *Silent Phase*. Specifically, we reverse-engineer the APK files and explore the source codes to figure out what the accessed sensor data are used for. Then we compare the usages with the description and privacy policy of applications.

6.3.1 Sensor Access Patterns and Third-Party Libraries

After reverse-engineering the APK files, we find that most of them access location sensors by third-party ad libraries or analytic libraries. In order to trace the data flow, we first scan the source codes for the sensor-related API and record the third-party package names if any. Next, combining with the access records and the applications' basic information we collected beforehand, we judge if there is any abuse or misuse of sensor data.

Among all applications in *Wandoujia*, there are 70 applications which accessed sensors during *Silent Phase*, and 53 of them accessed in a relatively high frequency. Because part of the applications applied anti-reverse engineering technologies or the tools we used to reverse-engineer exist deficiencies, we can only get the source codes of 42 applications. Most of the applications accessed sensors according to an obvious pattern. For example, there are 16 applications accessed location service every 4,850 seconds. This is because all of them contain a third-party library named *cn.jpush.android*, which includes the codes that will call the location sensors related services. In other cases, there are 6 applications contain the location sensor calling methods both in themselves and the third-party libraries they employ, and only one application's sensor related codes are not contained in third-party libraries. Therefore, the overall percentage of sensor accessing in third-party libraries is 83.33 percent.

For *360* market, there are 48 applications accessed sensors frequently during *Silent Phase*. After successfully reverse-engineering 28 of them, we find that, similarly to the case in *Wandoujia*, 6 applications present an accessing pattern with a fixed accessing interval of 4,850 seconds, which is caused by *cn.jpush.android* as well. 75 percent of the applications are found containing the sensor accessing codes in their third-party libraries. This phenomenon further verifies the fact that third-party libraries are blamed for the frequent suspicious accessing of sensors during *Silent Phase*.

While in *Google Play Store*, we do not find a uniform pattern since the accesses during *Silent Phase* are fewer than

TABLE 12
Top Third-Party Libraries Which Appears the Most Frequently in Our Experiments

Market	Package Name of Third-Party Libraries	Times
Wandoujia	*cn.jpush.android*	22
	com.tencent.map	14
	com.baidu.location	9
	com.qq.e.comm.managers.status	8
	com.loc	7
	com.amap.api.location	7
	com.aps	7
	com.alipay.mobilesecuritysdk.model	6
	com.tencent.mm.sdk.platformtools	6
360	*cn.jpush.android*	13
	com.baidu.location	12
	com.tencent.map	7
Google Play Store	*com.flurry.sdk*	9
	gms	9

that of the above two markets. There are 34 applications accessed sensors during *Silent Phase* and only 18 of them accessed in a relatively high frequency. However, we still find that 12 applications accessed sensors because of the related codes in third-party libraries' packages, which account for 66.67 percent of the total. And the other 6 applications contain the sensor accessing codes both in themselves and their containing third-party libraries.

From all the studies above, we find that 77.27 percent of the accessing records are caused by the third-party libraries. Table 12 shows the top third-party libraries which are used in our experiments. It reasonably explains the reason why the applications belongs to *Finance* and other categories, which seems to have no need to access sensors frequently, would generate so many accessing records in our experiment: although the applications themselves may have no intention to fetch any sensor data while running in the background, the third-party libraries call the sensor related service, which disobey the original motivation of the applications' developers. This possibility of sensor usages in third-party libraries can hardly be noticed by the developers and thus they would not mentioned in applications' descriptions, which would mislead the users and bring about the risk of privacy leakage. As a result, we argue that a responsible application should not only give a detailed description of the application's functionalities, but also list the sensor permissions required by both the application and its contained third-party libraries if any. Moreover, the description should clarify that which kind of sensor will be used under what circumstances, thus to eliminate the worries about privacy information leakage from users.

6.3.2 *Suspicious Case Study*

We further traced the data flow and summarized the concrete usage scenarios of the sensor data. Basically, we focus on the applications themselves how to use the data rather than the third-party libraries. After the analysis, we summarize the usage scenarios of location sensors' data as following:

1) Sending the collected sensor data to a specific server, but the further use can not be traced.
2) Using the collected data as a keyword for other information, e.g., pulling down the local weather information after accessing the location sensor and getting the location information.
3) Storing the collected data locally on device for the possible future use.
4) Exporting the sensor data to the system logs.
5) Doing nothing at all.

TABLE 13
The Application Packages Which Access the Location Sensor Data During *Silent Phase* and Their Corresponding Usage Scenario

package name	scenario	package name	scenario
com.letv.android.client	4	com.nd.android.pandahome2	5
cn.ledongli.ldl	3	viva.reader	1
com.jsmcc	2	com.cmcm.whatscall	1
com.vlocker.locker	2	com.handmark.expressweather	3
com.cleanmaster.mguard_cn	3	com.pingenie.screenlocker	2

Based on the different usage scenarios, we categorize the application packages according to their usage in Table 13. Besides, we also have an insight into the usages of standard sensor data. Basically, we can divide the ways they are used into as following: (1) Testing the device's rotation angle; (2) Testing the device's shaking; (3) Testing the brightness surrounding. And a detailed result is shown in Table 14.

At last, we investigated all the descriptions displayed on the application markets of applications mentioned above, and found that none of them notices the possible sensor usage. Thus we want to emphasize the importance of application's description, and insist that the developers should be responsible for clarifying the third-party libraries used in the application and which of them would possibly cause the use of sensors.

7 Discussion

7.1 Coverage

senDroid can implement sensing audit by (1) intercepting the IPC between applications and services or (2) intercepting the communication with sensor drivers. Considered in terms of generality, all four categories sensors can be audited by the first approach because the IPC mechanisms are uniform in different version of Android. The first approach can be detoured by attacks that directly access sensor drivers without communicating with the sensor service. So with respect to robustness, *senDroid* can intercept the communication with sensor drivers to defense the attacks or detect suspicious accesses to sensors.

TABLE 14
The Application Packages Which Access the Standard Sensor Data During *Silent Phase* and Their Corresponding Usage Scenario

package name	usage scenario
com.cleanmaster.mguard_cn	3
cld.navi.mainframe	1
com.lashou.groupurchasing	2
com.taobao.ju.android	1
com.ubercab	1

The coverage of the second approach depends on the source code of sensor drivers we can analyze. For graphic sensors and audio sensors, Linux kernel provides standard device drivers. Thus, by making the interception according to the standard invocation mode, *senDroid* is capable of auditing all accesses to graphic and audio sensors. For location sensors, with no device driver provided by Linux kernel, *senDroid* only supports sensing audit on devices that use Qualcomm GPS driver. Because the driver of standard sensors on Nexus 4 is close-source, *senDroid* only implements the sensing audit by intercepting the IPC between applications and services. We argue that, with the support of manufacturers of sensors, the location sensors and standard sensors can be audited by intercepting the communication with sensor drivers.

7.2 Suspicious Usage Patterns

senDroid provides users with the detailed and visual sensing usage reports. Users can infer from the usage reports which stealthy application is accessing sensors at an unexpected time or with an abnormal data size. For professional users, the reports of *senDroid* may offer more technical details, and help the professional users find more suspicious accesses to sensors. However, *senDroid* does not support the suspicious usage pattern recognition now. This absence could bring up a burden on common users, especially, when the sensing relevant applications are widely used.

A possible solution is to collect plenty of sensor data usage patterns of malwares. Then we can mine classification rules by applying machine learning on the usage patterns. Based on the suspicious usage patterns, we can then improve *Sensing Monitor* to be an application which can automatically identify the suspicious usage of running applications, and alert users if the alert rules are applied.

7.3 Bypassing *senDroid*

senDroid implements the sensing audit by intercepting the data flow and the interception relies on substituting function pointers in the ELF file of a target process. A malicious application can apply the similar interception to bypass *senDroid* either by substituting the sensor-related system calls for its own implementation, or by breaking down *senDroid* completely.

However, it is challenging for the attacker because the malicious application must call `ptrace` to attach to the target process before function substitution. So if *senDroid* is attached to the target process before the attacker, according to the documentation of `ptrace` [33], the later attaching will cause error and thus fail. *senDroid* is attached to target processes as soon as the system is launched. Even when attacker makes the attachment before *senDroid*, we can be informed of the abnormal behavior and give the user a warning.

8 Related Work

Hooking is a technique for inserting codes into a system call for alteration, and it can be used to intercept the applications' requests, thus to realize the sensor-related behavior check. Xu et al. developed *Aurasium* in [27]. *Aurasium* can keep on monitoring any security or privacy violations in Android OS. Basically, *Aurasium* realizes the enforcement of its security policies by hooking into applications processes. *FireDroid* [34] is another work that relies on hooks. It intercepts the system calls to identify if an application is executing dangerous actions at runtime. Similarly, *FireDroid* performs security checks on applications and enforces security policies. *DeepDroid* [35], a dynamic enterprise security policy enforcement scheme on Android devices, also dynamically hooks system processes in order to find details of applications' requests for a fine-grained access control. *Boxify* [36] presents a concept for full-fledged app sandboxing on stock Android and it also aims at enforcing established security policies. Although the core technique used in these works are similar to ours, however, no matter in *Aurasium*, *FireDroid*, *DeepDroid* or *Boxify*, they aim at realizing a policy-based security, which means they design the system with hooks in order to apply some specific security policies to prevent users from attacks, while our work is different from others by focusing on auditing sensor access in Android system-wide.

Besides hooking, another technique is to modify the existing Android sensor framework to intercept the sensor data flow. Xu et al. [37] proposed a sensor management framework, called *SemaDroid*, based on the *SemaHook*s, which are not real hooks but are codes embedded within the existing components in the Android framework. *SemaDroid* provides the users with capacity of monitoring the sensor usage of installed applications and also provides a fine-grained and context-aware access control. Basically, *SemaDroid* focuses on supporting of context-aware and quality-of-sensing based access control policies though it offers the possibility of auditing the sensor usage by giving a sensor usage report as an individual application as well.

ipShield [38] is a framework that monitors sensor accessed by an application and assesses the privacy risk of the sensor access. Besides, it also gives recommendations of sensor configurations to users and supports sensor related access control actions. *Scippa* [39], an extension to the Android IPC mechanism, provides provenance information required to effectively prevent recent attacks such as confused deputy attacks. Heuser et al. [40] proposed the Android Security Modules (ASM) framework, which provides a programmable interface for defining new reference monitors for Android. Particular reference monitor can be developed to monitor sensor access. These designs can be bypassed in the access mode of JNI, where the accesses to sensors do not pass the Android framework layer. Different from these works, *senDroid* implements the interception at the device driver layer without any modification to the Android framework. So *senDroid* can be widely deployed and is capable of detecting potential attacks that bypass the Android framework. *senDroid* can not only detect which application is accessing which sensor but also quantify how many data the application accessed. Furthermore, the hook-based method can audit applications even when they bypass the Android framework via JNI invocations. The implementation and evaluation show that *senDroid* is effective and efficient to audit sensing in the Android platform.

Enck et al. [5] proposed a dynamic taint tracking and analysis system, named *TaintDroid*, that is capable of simultaneously tracking multiple sources of sensitive data. *DroidTrack* is a method proposed in [41] for tracking and visualizing the transmission of privacy information and preventing its leakage. *AppIntent* [42], a framework analyzes data diffusion to help an analyst to determine whether the data diffusion is user intended or not. These mechanisms can track the target data, including personal privacy. The tracking reports of them are very limited while *senDroid* reveals more details of the sensing operations.

Wijesekeraet al. [43] did a field study to find how often applications access protected resources when users are not expecting it on Android platform and they hooked the permission-checking APIs in data collection step. However, in their study, they focus on the permissions or resources of connectivity, location, view, and so on. In *senDroid*, we implement a meticulous study focusing on sensors on the Android platform.

9 Conclusion and Future Work

This paper proposes *senDroid*, which may be used to monitor and analyze the sensing operations in the Android platform. To the best of our knowledge, it is the first work to design a tool to audit the sensing in the Android platform without the changes of the source codes of the Android framework. *senDroid* leverages a hook-based method to implement the interception of the sensor related API calls. According to the results of our conducted experiments, *senDroid* can efficiently gather the data of all four categories of sensors in the Android platform, i.e., graphic sensors, audio sensors, location sensors, and standard sensors, with high accuracy. In addition, *senDroid* can work even when suspicious or malicious codes are dynamically loaded from server sides or bypass the middleware of Android via JNI calls. Next, *senDroid* can monitor the suspicious behaviors where an application extracts the graphic data from the preview frames of a camera to silently take photos. This behavior can bypass the alert of the shutter voice which is mandatorily open in some countries, such as China. The performance report shows that the [0.04-8.05] percent overheads for different operations are promising. Finally, our empirical study on applications from real markets shows that it is very high-frequent for popular applications in *Wandoujia*, *360* and *Google Play Store* to access sensors. We also find that many applications access location sensors and standard sensors when applications are running in the background and third-party libraries are blamed for the continuous accesses, but the developers do not declare the usage in the description or privacy policy of the application. To the best of our knowledge, it is also the first empirical study on the dynamic usages of sensors of Android applications.

In our future work, we plan to conduct more experiments for a large scale applications to discover more cases of suspicious and malicious usages of sensors. In addition, we will improve our analysis tool to automatically report the malicious usages based on our gathered data and other relevant data of applications, such as application descriptions. Last but not least, we will support the further sensing audit under the supports of device manufactures.

Acknowledgments

This paper is supported by NSFC (Grant No. 61572136, 61472358, 61370080), the National Program on Key Basic Research (2015CB358800), and the Shanghai Innovation Action Project (Grant No. 16DZ1100200, 18511103600). We thanks all anonymous reviewers for their insightful comments. We are now sharing all source codes of *senDroid* on GitHub (https://github.com/letitb/senDroid).

References

[1] L. Atzoria, A. Ierab, and G. Morabitoc, "The internet of things: A survey," *Comput. Netw.*, vol. 54, pp. 2787–2805, 2010.
[2] R. Liu, "Alipay dismisses accusation it violated user-privacy by snapping photos." [Online]. Available: http://www.allchinatech.com/alipaydismisses-accusation-it-violated-user-privacy-by-snappingphotos/, Accessed on: 2016.
[3] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion," in *Proc. 6th USENIX Workshop Hot Topics Security*, 2011, vol. 11, pp. 9–9.
[4] Y. Michalevsky, D. Boneh, and G. Nakibly, "GyropPhone: Recognizing speech from gyroscope signals," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 1053–1067.
[5] W. Enck, et al., "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014, Art. no. 5.
[6] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang, "Accomplice: Location inference using accelerometers on smartphones," in *Proc. 4th Int. Conf. Commun. Syst. Netw.*, 2012, pp. 1–9.
[7] S.-W. Lee and K. Mase, "Activity and location recognition using wearable sensors," *IEEE Pervasive Comput.*, vol. 1, no. 3, pp. 24–32, 2002.
[8] P. Mohan, V. N. Padmanabhan, and R. Ramjee, "Nericell: Rich monitoring of road and traffic conditions using mobile smartphones," in *Proc. 6th ACM Conf. Embedded Netw. Sensor Syst.*, 2008, pp. 323–336.
[9] T. Watanabe, M. Akiyama, and T. Mori, "Routedetector: Sensor-based positioning system that exploits spatio-temporal regularity of human mobilty," in *Proc. Usenix Conf. Offensive Technol.*, 2015, p. 6.
[10] D. Currie, "Shedding some light on voice authentication," 2009.
[11] A. Al-Haiqi, M. Ismail, and R. Nordin, "On the best sensor for keystrokes inference attack on android," *Procedia Technol.*, vol. 11, pp. 989–995, 2013.
[12] R. Spreitzer, "Pin skimming: Exploiting the ambient-light sensor in mobile devices," in *Proc. 4th ACM Workshop Security Privacy Smartphones Mobile Devices*, 2014, pp. 51–62.
[13] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, "Mobile device identification via sensor fingerprinting," *CoRR*, vol. abs/1408.1416, 2014, [Online]. Available: http://arxiv.org/abs/1408.1416
[14] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi, "AccelPrint: Imperfections of accelerometers make smartphones trackable," in *Network and Distributed System Security Symposium*. New York, NY, USA: Citeseer, 2014.
[15] H. Wang, D. Lymberopoulos, and J. Liu, "Sensor-based user authentication," in *Wireless Sensor Networks*. Berlin, Germany: Springer, 2015, pp. 168–185.
[16] T. Schreiber, "Android binder," *A Shorter, More General Work, but Good for an Overview Binder*. 2011. [Online]. Available: http://www.nds. rub.de/media/attachments/files/2012/03/binder. pdf
[17] Android, "Android interface and architecture." [Online]. Available: https://source.android.com/devices/, Accessed on: 2015
[18] B. Nguyen, "Linux dictionary," 2003.
[19] eLinux.org, "Executable and linkable format (elf)." [Online]. Available: http://elinux.org/Executable_and_Linkable_Format_(ELF), Accessed on: 2015
[20] E. Bendersky, "Position independent code (PIC) in shared libraries." [Online]. Available: http://eli.thegreenplace.net/2011/11/03/positionindependent-code-pic-in-shared-libraries/, Accessed on: 2015
[21] J. Shewmaker, "Analyzing DLL injection," *GSM Presentation*, 2006.
[22] P. Padala, "Playing with ptrace, part II," *Linux J*, vol. 104, 2002, Art. no. 4.
[23] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proc. 5th ACM Conf. Security Privacy Wireless Mobile Netw.*, 2012, pp. 113–124.
[24] R. Schlegel, K. Zhang, X.-Y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proc. 2nd Netw. Distrib. Syst. Security Symp.*, 2011, vol. 11, pp. 17–33.

[25] Z. Zhang, P. Liu, J. Xiang, J. Jing, and L. Lei, "How your phone camera can be used to stealthily spy on you: Transplantation attacks against android camera service," in *Proc. 5th ACM Conf. Data Appl. Security Privacy*, 2015, pp. 99–110.

[26] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. USENIX Security Symp.*, 2011, vol. 2, Art. no. 2.

[27] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proc. Presented Part 21st USENIX Security Symp.*, 2012, pp. 539–552.

[28] typcn, "typcn on twitter." [Online]. Available: https://twitter.com/typcn_com/status/701706390218231808, Accessed On: 2016–05-10.

[29] bunnyblue, "Acdd," [Online]. Available: https://github.com/bunnyblue/ACDD, accessed: 10-May-2016

[30] AnTuTu, "Antutu benchmark 3d- android apps on google play." [Online]. Available: https://play.google.com/store/apps/details?id=com.antutu.benchmark.full, Accessed on: 10-May-2016

[31] Android, "Audio capture." [Online]. Available: https://developer.android.com/guide/topics/media/audio-capture.html, Accessed on: 2016

[32] Android, "Sensors overview." [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_overview.html#sensors-practices, Accessed on: 2016

[33] M. Kerrisk, "ptrace(2)." [Online]. Available: http://man7.org/linux/man-pages/man2/ptrace.2.html, Accessed on: 2016

[34] G. Russello, A. B. Jimenez, H. Naderi, and V. D. M. Wannes, "Firedroid: Hardening security in almost-stock android," in *Proc. Comput. Security Appl. Conf.*, 2013, pp. 319–328.

[35] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices," in *22nd Ann. Netw. Distrib. Syst. Secur. Symp.*, NDSS 2015, San Diego, California, USA, Feb. 8–11, 2015.

[36] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. Von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *Proc. Usenix Security Symp.*, 2015, pp. 691–706.

[37] Z. Xu and S. Zhu, "SemaDroid: A privacy-aware sensor management framework for smartphones," in *Proc. 5th ACM Conf. Data Appl. Security Privacy*, 2015, pp. 61–72.

[38] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava, "ipShield: A framework for enforcing context-aware privacy," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2014, pp. 143–156.

[39] M. Backes, S. Bugiel, and S. Gerling, "Scippa: System-centric IPC provenance on android," in *Proc. Comput. Security Appl. Conf.*, 2014, pp. 36–45.

[40] S. Heuser, A. Nadkarni, W. Enck, and A. R. Sadeghi, "ASM: A programmable interface for extending android security," in *Proc. 23rd USENIX Conf. Security Symp.* 2014, pp. 1005–1019.

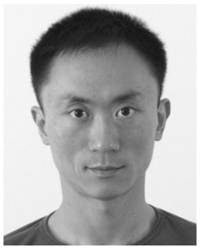
[41] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi, "DroidTrack: Tracking and visualizing information diffusion for preventing information leakage on android," *J. Internet Serv. Inf. Security*, vol. 4, no. 2, pp. 55–69, 2014.

[42] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2013, pp. 1043–1054.

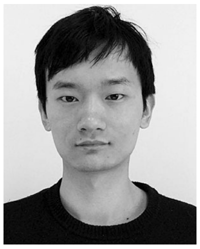
[43] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proc. Usenix Conf. Security Symp.*, 2015, pp. 499–514.

Weili Han (M'08) is a full professor in the Software School at Fudan University. His research interests are mainly in the fields of data systems security, access control, digital identity management. He is now a member of the ACM, SIGSAC, IEEE, and CCF. He received his Ph.D. degree at Zhejiang University in 2003. Then, he joined the faculty of the Software School at Fudan University. From 2008 to 2009, he visited Purdue University as a visiting professor funded by China Scholarship Council and Purdue University. He serves in several leading conferences and journals as PC members, reviewers, and an associate editor. He is a member of the IEEE.

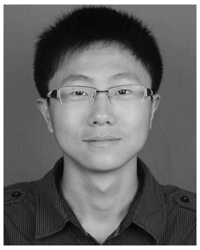
Chang Cao is a graduate student majored in Computer Software and Theory at Fudan University. She received her B.S. degree from Fudan University in 2016. She is currently a member of the Laboratory for Data Analytics and Security. Her research interest mainly includes sensor-related access control on mobile devices.



Hao Chen is a full professor in the Department of Computer Science at the University of California, Davis. He received his Ph.D. degree at the Computer Science Division at the University of California, Berkeley, and his BS and MS from Southeast University. His research interests are computer security, machine learning, and mobile computing.



Dong Li is currently a software engineer in Works Applications Co., Ltd. He received his B.S. degree in Computer Science from Beijing Institute of Technology in 2014 and the M.S. degree in Computer Software and Theory from Fudan University in 2017. His research interests are mainly in the fields of access control on Android.



Zheran Fang is currently a software engineer at Microsoft. He received his M. S. degree in Computer Science from Fudan University. He was a member of the Laboratory of Cryptography and Information Security, Software School, Fudan University. His research interests mainly included information security and policy based management.

Wenyuan Xu (M'07) is a full professor in the College of Electrical Engineering at Zhejiang University. She received her B.S. degree in Electrical Engineering from Zhejiang University in 1998, her M.S. degree in Computer Science and Engineering from Zhejiang University in 2001, and the Ph.D. degree in Electrical and Computer Engineering from Rutgers University in 2007. Her research interests include wireless networking, smart systems security, and IoT security. Dr. Xu received the NSF Career Award in 2009 and was selected as a young professional of the thousand talents plan in China in 2012. She was granted tenure (an associate professor) in the Department of Computer Science and Engineering at the University of South Carolina in the U.S. She has served on the technical program committees for several IEEE/ACM conferences on wireless networking and security. She has published over 60 papers and her papers have been cited over 3000 times (Google Scholar). She is a member of the IEEE.

X. Sean Wang is a full professor in the School of Compute Science at Fudan University, Shanghai, China. He received his Ph.D. degree in Computer Science from the University of Southern California in 1992. Before joining Fudan University in 2011, he was the Dorothean Chair Professor in Computer Science at the University of Vermont. His research interests include data systems and data security. He is a member of ACM and senior member IEEE.