Towards Dynamically Monitoring Android Applications on Non-rooted Devices in the Wild

Xiaoxiao Tang
Singapore Management University
xxtang.2013@smu.edu.sg

Yan Lin
Singapore Management University
yanlin.2016@smu.edu.sg

Daoyuan Wu
Singapore Management University
dywu.2015@smu.edu.sg

Debin Gao
Singapore Management University
dbgao@smu.edu.sg

ABSTRACT

Dynamic analysis is an important technique to reveal sensitive behavior of Android apps. Current works require access to the code-level and system-level events (e.g., API calls and system calls) triggered by the running apps and consequently they can only be conducted on in-lab running environments (e.g., emulators and modified OS). The strict requirement of running environment hinders their deployment in scale and makes them vulnerable to anti-analysis techniques. Furthermore, current dynamic analysis of Android apps exploits input generators to invoke app behavior, which, however, cannot provide sufficient code coverage.

We propose to dynamically analyze app behavior on non-rooted devices used by the public so that it is possible to analyze dynamically in scale without input generators. By doing so, we also maximize the code coverage since the app behavior is invoked by real users of the apps. To achieve such a goal, we build UpDroid, a system for detecting sensitive behavior without modifying Android OS, rooting the device, or leveraging emulators. UpDroid detects sensitive events by monitoring the changing of public resources on the device, instead of accessing low-level events that require rooting or system modification. To identify the apps that trigger the detected events, UpDroid formulates the identification as a ranking problem and adopts learning to rank technique to solve it. Our experimental results demonstrate that UpDroid can successfully detect the use of 15 out of 26 permissions that are labeled dangerous in the official Android documentation. We also compare UpDroid with API hooking which can theoretically capture all sensitive behavior but requires root permission and system modifications. Results show that UpDroid can still achieve 70% coverage of API hooking even without root permission or any system modifications.

WiSec'18, Stockholm, Sweden

DOI: 10.475/123_4

1 INTRODUCTION

Android has been the most popular mobile system which occupies over 85% market share in Q1 2017 [18]. Along with the popularity, the Android community also faces various threats, such as malware[41], pirated apps [40] and so on. One of the most important mitigation techniques for these threats is the effective and precise dynamic analysis to reveal the underlying sensitive behavior of apps.

Most of existing techniques conduct analysis on emulators or modified systems. Analyzing under these environments requires input generation tools [16, 22, 33] to automatically execute the target apps. However, most input generators can only provide a random series of events, e.g., touching on the screen, to mimic real users' behavior. The random behavior generated by these tools can hardly match the pattern of the real app usage to successfully invoke certain functionalities, e.g., registration. Hence, input generators cannot provide as wide code coverage as humans. Choudhary et al. compared several popular monkey tools and found that the best coverage that these tools can reach is 40% [9]. Meanwhile, anti-analysis techniques [19, 30] allow apps to recognize the running environment and hide their sensitive behavior accordingly. For example, apps can detect whether the running environment is emulated based on the GPS info or IMEI number. Moreover, anti-analysis techniques can choose to trigger sensitive behavior only under specific circumstances that have no dependencies on program inputs, e.g., after receiving an SMS, at a particular time slot, or when receiving a remote command [31]. Both the insufficient code coverage of the input generators and the anti-analysis features of the target apps hinder existing dynamic analysis from invoking the potential behavior of them. Theoretically, to enable large-scale deployment and evade anti-analysis techniques, the optimal solution is to conduct the analysis on devices used by the general public. In this paper, we study to what extent dynamic analysis can be applied to non-rooted and unmodified devices.

Dynamically analyzing apps' behavior on such devices is challenging. Previous tools [5, 7, 11, 28, 29, 36] adopt API tracing, system call tracing and so on, to infer the underlying behavior of the apps. As low-level information (e.g., API call or system call) commonly used by previous works is not accessible on non-rooted devices, these techniques cannot be applied to devices used by the public. To deal with this problem, we propose a system called UpDroid. Instead of logging low-level events, we monitor the state

changing of different types of public resources on the target device. The changes convey information about the sensitive behavior of the apps. For example, we can monitor message sending behavior by detecting the newly added rows of the content provider `content://sms`. The changing event corresponds to behavior that has been *successfully* performed on the devices, which is different from detecting *attempts* of actions through tracing API calls. Unlike existing works which can hook into the apps, monitoring the state changes of public resources brings another challenge – identifying the apps that trigger the monitored events. Hence, we use machine learning techniques to build a model for identifying the apps at runtime.

UpDroid can monitor various events including making phone calls, accessing the camera, reading/writing files and so on. It achieves around 80% precision in identifying the apps that trigger the observed events. We compare UpDroid with the traditional API hooking to study how far UpDroid can go in covering different types of behavior and how it is different from the traditional hooking method. Experimental results demonstrate that the events UpDroid can capture cover 15 out of 26 dangerous permissions, while API hooking covers 21. The permissions covered by UpDroid contain the popular ones used by both malware and benign apps. From tests on several popular apps, we observe that UpDroid detects the result-based events and API hooking misses some because of the incompleteness of the sensitive API list.

The main contributions of this work are as follows.

- We propose a dynamic analysis system named UpDroid that is applicable to unmodified and non-rooted devices used by the public.
- We propose several methods for monitoring different types of sensitive behavior based on the state changing of public resources on the devices.
- We propose to use a machine learning technique – learning to rank, to establish the relationship between running apps and the detected events. This method addresses the challenge of app identification on unmodified devices.
- We compare UpDroid with the traditional API tracing method. The result shows that UpDroid can handle most of the cases that API tracing can handle and stands out in revealing the behavior that has been successfully performed.

The rest of the paper is organized as follows: Section 2 introduces the background knowledge and motivation of this work. Section 3 presents the framework of UpDroid. Section 4 and Section 5 discuss the detailed techniques. Section 6 presents the comparison between UpDroid and API hooking. Section 7 presents the capability analysis of UpDroid. Section 8 discusses the related work. Finally, Section 9 concludes this paper with future directions.

2 BACKGROUND AND MOTIVATION

In this section, we introduce some background information for this work and present the current state of sensitive behavior monitoring on Android to motivate this work.

2.1 Resources and Observers

Android has mature security protection mechanisms based on its permission model and the security features inherited from the Linux kernel. Guarded by these mechanisms, third-party apps have limited access to the static and runtime resources of the device. Normally, only with legal permission declaring and requesting an app could access the protected resources and perform sensitive behavior. In this paper, we propose to monitor state changing of four categories of resources which are normally available for third-party apps to detect the sensitive behavior.

Content Provider. Content provider is an app component provided by Android for managing access to a structured set of data. It is often used to store users' personal information, such as SMS, call logs, contact information and so on [14]. Content provider encapsulates the data and provides mechanisms for security. With proper permission, third-party apps can access the content providers that are open to external apps. Various functionalities are implemented with content providers, e.g., the default app for sending and receiving SMS uses content provider to store the SMS logs. Android provides the `ContentObserver` API for receiving callbacks of changes to a content provider to monitor the content provider events. For example, a malware named HongTouTou uses this API to monitor the SMS content provider and delete particular SMS according to the changes [37].

External Storage. The file system of Android inherits that of the Linux kernel. The files are protected with read/write/execute permission for each user. Therefore, on non-rooted devices, we can only monitor the files or directories which are readable to third-party apps. For example, we cannot monitor system files under `/data/` directory, since the external apps don't have the read permission. External storage (known as the `/sdcard` directory) is a platform-specific file system module on Android, which is public to third-party apps . To access external storage, apps always need permission `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`. In this work, we focus on monitoring external storage directory. Previous works use `FileObserver` to notify the file system changes [17].

Interrupt Statistics. The logs of the interrupts raised to the kernel are also readable to third-party apps through a virtual file interface – `/proc/interrupts`. With this interface, users can obtain information about how many interrupts have been received by the CPU since booting. Previous work [12] uses this interface to infer user's sensitive information, e.g., unlock pattern. We can use this interface to observe the use of different resources on the device, e.g., the camera, the Bluetooth, the NFC and so on. More details can be found in Section 4.

Network. Network is another kind of resource for users to access during runtime. With `INTERNET` or `ACCESS_NETWORK_STATE` permission, apps can obtain the connection state, open URLs, or send/receive TCP/UDP packets. To monitor the network activities, previous work MopEye [34] leverages `VpnService` API to intercept all traffic initiated from apps on the devices. This API is designed for app developers to build VPN apps.

Although we can observe these resources to represent sensitive events on the device, identifying the apps that trigger the events is still a mystery. In this paper, we integrate these observers to build a monitor for capturing the sensitive events on the devices and use machine learning techniques to identify the initiator of them.

2.2 Motivation

Table 1 lists existing works about dynamic analysis of sensitive behavior. We can observe that they need to use API calls, system calls, and other low-level events to reveal the underlying behavior of the target apps. For example, CopperDroid [29] observes and dissects the system calls made by an app to reconstruct the behavior, e.g., file operations. A majority of these works are based on in-lab running environment, including VM (Virtual Machine)-based emulators, modified OS/Android internals, and rooted devices.

Table 1: Existing tools for analyzing sensitive behavior of Android apps

Tool	Platform	Features
DroidScope [36]	QEMU based Emulator	API call, system call, Dalvik instruction and so on.
CopperDroid [29]	QEMU based emulator	API constructed from system call
VetDroid [39]	Modified system	API call
M. Karami et al. [20]	Any platform with instrumented App	System call
DroidBox [21]	Modified system	API call

Relying on the in-lab running environment, previous work requires input generation tools [16, 22] to automatically run the target apps. However, the event series generated by these tools cannot match the logic of mobile apps which is usually complicated, e.g., most apps require registration following strict commands. Choudhary et al. present that the maximum coverage of popular input generator tools is only 40% even with sufficient time for running the apps [9]. Our intuition is that humans may be more successful in invoking the relevant functionalities of apps, and thus can achieve better code coverage with enough time and a large number of users. On another hand, app developers, especially those who design malware, would not prevent their apps' behavior from being triggered under real execution environments. Hence, deploying dynamic analysis to public users for crowdsourcing solves the code coverage problem. Besides the coverage problem, running on emulators cannot analyze some environment sensitive apps. Petsas et al. [26] proposed a range of techniques to evade dynamic analysis in the emulated Android environment. With these techniques, apps can bypass the analysis of the tools, e.g., CopperDroid and DroidScope.

In this paper, we introduce our dynamic analysis system named UpDroid. It gathers data from users' daily running traces and generates sensitive behavior reports for the apps.

3 SYSTEM OVERVIEW

Figure 1 shows the framework of UpDroid, which consists of two major components: the monitoring module on the users' devices and the analysis module on the server side. We place two monitors on Android devices to monitor sensitive behavior (e.g., accessing the camera) and collect runtime status (e.g., CPU usage) of running apps. The data collected by both monitors is logged with time stamps. The event monitor detects changes to resources which can be accessed by third-party apps, e.g., the file system, to reveal the apps' behavior. We choose to use these changes to represent the sensitive behavior as low-level events are not accessible on non-rooted devices. However, we cannot identify the initiating app for the detected events without penetrating to apps or the systems. Hence, in the analysis module, we build an app identification model with machine learning techniques to distinguish the initiating app from all running apps. We use learning to rank to train the model with data from real users as presented in Section 5. We take sensitive events and the corresponding runtime status of the apps as inputs to identify the initiating apps for the monitored events and generate behavior reports for the apps. In the following two sections, we will present the technical details of the event monitoring and how we identify the initiating app for each event.

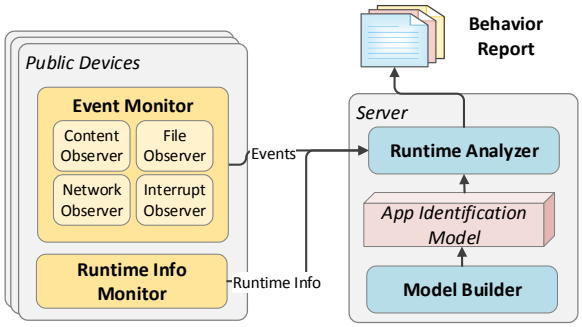

Figure 1: Framework of the sensitive behavior monitoring system - UpDroid.

4 EVENT MONITORING

This section describes how UpDroid monitors sensitive events without penetrating to either the apps or the Android internals on non-rooted phones. To reveal the behavior of the apps, UpDroid passively captures the events triggered by sensitive behavior. Figure 2 presents four types of sensitive behavior that can be monitored by UpDroid. The behavior is categorized by the resources, e.g., the file system, it manipulates. For example, accessing the camera raises a particular interrupt, so it belongs to the interrupt-based behavior. We use different methods to monitor different categories of behavior.

4.1 Content Observer

UpDroid uses the `ContentObserver` API to capture the behavior that changes content providers. The method `registerContentObserver(Uri uri, boolean notifyForDescendants, ContentObserver observer)` is used to register a content observer with the corresponding URI, e.g., `content://sms` for observing SMS content. When an event is detected, the `onChange()` method will be triggered. The `ContentObserver` only reports whether a content provider is changed, but not what has been changed. Hence, we log the monitored content provider and compare the updated provider with that at a previous timestamp after receiving a change notification, in order to get detailed information for inferring the apps' behavior. For example, row adding of SMS content provider with entry `type=0` represents sending out an SMS, and row adding with entry `type=1` represents receiving an SMS. The entries for each

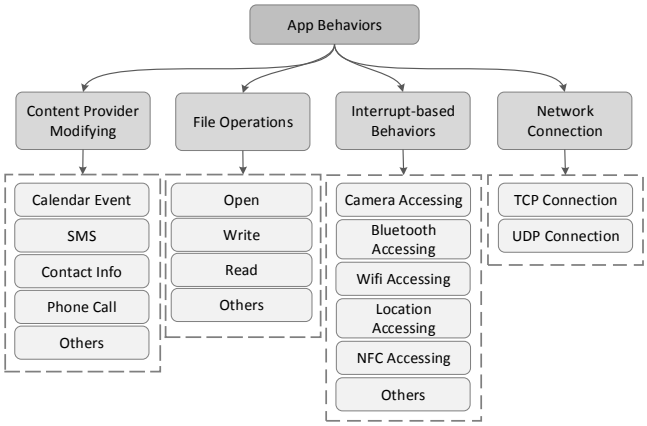

Figure 2: Different categories of behavior that UpDroid can monitor

row also provide information, such as when the SMS is sent and the recipient of the SMS. To find all observable content providers, we use `PackageManager` to list all providers which can be accessed by external apps. For each provider, we query the corresponding database to find all table names which is also used as the path prefixes of the URIs of providers. From Android 6.0.1, we find 21 system provided content providers. Axplorer [3] can also identify the system content providers that are protected by permissions, but it does not filter out the ones that can only be accessed by the system apps. Theoretically, UpDroid can monitor all content providers with the required permissions. However, due to the overhead of logging and comparing the content providers, UpDroid only observes four of the most common and significant content providers, including SMS, call log, contacts and calendar events.

4.2 File Observer

To monitor events related to the file system, UpDroid uses `FileObserver` API to monitor the files and directories on the external storage. This API is provided by Android to capture changes to a single file or a directory. Event masks (e.g., `CREATE`, `DELETE` and `MODIFY`) are used to specify what kind of operation has been performed on the monitored file or directory. A complete list of the event masks can be found in Android API reference [15]. The `onEvent()` method will be triggered when an event to the file or directory is observed. Since this API only supports single file or directory monitoring, we recursively traverse the monitored directory and register file observer for each file or directory under it. Similar to `ContentObserver`, `FileObserver` only reports the events but not the changing content. For example, `FileObserver` does not report how the file is modified when it captures a `MODIFY` event. Backing up the target directory is a possible solution to get detailed information about the events, but this may bring in too much space and runtime overhead. Hence, UpDroid only observes different types of events to the files in the external storage and ignores the detailed changes to them.

4.3 Interrupt Observer

A novel method we propose for observing events is to sample the interrupts and monitor the changes of interrupt numbers. Android inherits the interrupt mechanism of Linux. Interrupt represents the situation where CPU interrupts the running program to handle a request raised by an external hardware device. When the devices (e.g., camera, Bluetooth and temperature sensor) detect physical events, they raise interrupt requests. Then, the programmable interrupt controller (PIC) will process these requests and send them to the CPU. The CPU will finally respond to the interrupt requests. Each specific interrupt will be registered to the system with a unique Interrupt Request Line (IRQ) number, through which devices can pass the interrupt to the processor. The virtual file `/proc/interrupts` provides the interrupt request lines claimed by the devices. Each line shows the unique IRQ number, the number of interrupts handled by each CPU, the PIC, and the device name.

To identify the events from the number of interrupts, we sample the `/proc/interrupts` file each 100ms and compare it with the previous sampling. Since most hardware devices have a corresponding IRQ line, we can infer the running status of hardware through monitoring the changes to the numbers of the interrupts. The increases of the interrupts represent the sensitive behaviors. For example, accessing the camera increases the number of interrupt number 83 on Nexus 6P. Using Bluetooth to send a file to another device will increase interrupt 503 continuously for a period. In UpDroid, we choose to monitor the following five common devices: camera, GPS, Bluetooth, NFC and video decoder.

Most device names shown in `/proc/interrupts` are coded with the hardware model names or abbreviations, thus are difficult to identify. For example, on HUAWEI Nexus 6P, `pn548` is the interrupt name of NFC and `atmel_mxt_ts` is for the touchscreen. Moreover, there are different IRQ lines with the same device name. For example, IRQ numbers 83 to 86 have the same device name `csid`, but only number 83 represents the camera device interrupt. Hence, the interrupt to hardware device mapping relies on the model names of the hardware devices which are difficult to obtain automatically.

Each mobile device has its own mapping between interrupts and hardware devices. Hence, we need to test the hardware devices and analyze the interrupt sampling to identify the interrupt mapping for each device model. We first analyzed several devices we already have, such as Nexus 6 and Nexus 6P. To cover other devices, we conduct a user study (named *interrupt study*) to obtain the mappings for them. Each UpDroid user needs to finish this study to get the interrupt mapping. The users need to perform certain operations to test the hardware devices on the phones. Some of the hardware devices can be tested automatically with proper programming. For example, we write a program to open the camera and take photos automatically. The others require the users' manual tests since the permission of these devices are very strict. For instance, NFC can only be manually turned on/off by the user for security consideration of Android. While the user is performing the tasks, the monitoring app samples the interrupts. By observing the changing pattern of the interrupts, we can identify the one that corresponds to the hardware devices. For each hardware, we need five traces for manually identifying the interrupt patterns.

From the recruited participants, we have identified interrupts for 19 Android devices.

4.4 Network Observer

UpDroid monitors the networking behavior through `VpnService` API, which leverages the TUN virtual network device to capture the TCP and UDP packets sent by the apps. In this work, we leverage MopEye's technique to identify the package initiators through the proc file `/proc/net/tcp6|tcp|udp|udp6` [34].

5 INITIATOR IDENTIFYING

The monitoring techniques presented in the previous section are not able to identify the app that triggers the detected event since the APIs used by the monitors are designed for observing the resources. Although MopEye [34] provides an intelligent solution for app identification on network events, it is not applicable to other events, e.g., interrupt-based events. Hence, UpDroid leverages a machine learning technique, learning to rank, to build an app identification model which is generic for all events. This model takes the detected event and the runtime information of the running apps as input, and ranks all the running apps to find out which is the one that initiates the detected event.

The overview of the model learning is presented in Figure 3. To get the ground truth for the model learning, we recruit Android users as the inspectors to identify the apps that trigger the detected events. After pre-processing the data from the inspectors and the monitors, we use the learning to rank technique to train the identification model with the feature vectors.

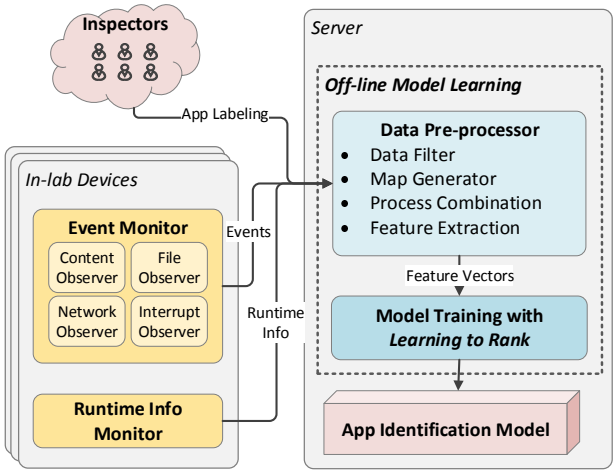

Figure 3: Overview of building the app identification model

5.1 App Status Monitoring

The runtime info monitor on the device collects information (e.g., CPU usage) about the running apps. We use the information as the feature of each app to infer whether it is the one that invokes the detected event. UpDroid uses the `ps` command to obtain the runtime information of the apps. It leverages the `/proc/stat` and `/proc/$PID/stat` interfaces to provide processes' runtime status, such as CPU usage, NICE value, virtual memory usage and so on. It allows third-party apps to access other processes' runtime info on most Android devices. We obtain the result from `ps` command from the monitoring app each 100ms. This time interval ensures both the quality of the data and the performance of the device.

One problem with using `ps` command is that the runtime status is for processes while identifying the process for an detected event is nearly impossible for users. For each occurred event, normal users can easily select the correct app that invokes the event, but can hardly tell which process without any knowledge about the app implementation. In this case, we consider *app* rather than *process* as the initiator of the detected events. We integrate the runtime status from processes from one app and get the ground truth of the initiators from the inspectors. More details of integrating runtime info from different processes can be found in Section 5.3.

5.2 Data Collecting

We collect the data for building the app identification model from the monitors presented in Section 4 and Section 5.1. The data contains the events and the runtime information of each app when an event occurs. The missing part is the ground truth that which one among all the running apps invokes the detected event. To get the ground truth, we recruit Android users to help to collect the data and label the app in real-time. We gather three types of information from the inspectors' devices - the events, the runtime info, and the initiating apps selected by the inspectors. We conduct a user study (named *initiator study*)[1] to collect and label the data. In this study, each participant will be asked to install our monitoring app published on Google Play and help to identify the apps at runtime. The monitor will capture the events and log the runtime app info. It would raise notifications to the participants (or inspectors) when it captures an event on the device. For each event, we provide the event type, the event content and the time when the event is captured to the participant. The participant responds to the notification and chooses the app that invokes the event based on the provided info. To check the integrity of the data, we propose the following policies to verify the participants' responses:

(1) The selected app should be on the list of the running apps.
(2) The selected app should have the permission for the detected event, e.g., the app chosen for a *camera* event needs to be granted with the `CAMERA` permission.
(3) The selection should be finished within ten mins after the notification, to make sure the user selects with a fresh memory.

We have recruited ten users since November 2017 to participate in the initiator study for data collection and labeling. The participants are Android phone users above 18 years old. Participants install the inspecting app on their own devices and identify initiators for at least 20 detected events. During the study, participants need to have at least ten apps they commonly use installed on their devices. We use the above policies to filter the responses from these participants. In total, we have collected 300 events with initiator

[1]Both the interrupt study and the *initiator study* have been approved by IRB in May 2017

identified. The initiators of these events correspond to 40 popular apps, e.g., Instagram and WhatsApp.

5.3 Data Pre-processing

From the monitors and the inspectors, we obtain the raw data for the analysis, including the detected events, runtime information of apps and the app identified by the inspectors. To get the labeled data, we pre-process the raw data in three steps. First, we obtain runtime app info which can represent the running apps' status for each detected event. We choose the nearest samples of runtime info before and after each detected event based on the timing info. Hence, for each event, we know the apps' current state and how it changes after the event occurs.

Table 2: Features for running apps and the process combination rules

Feature	Description	Type	Combination Rule
VSIZE	Size of virtual memory used	Integer	average
RSS	Resident set size	Integer	average
CPU	CPU usage	Integer	average
SCHED	Schedule of the process	Integer	average
PRIO	Priority	Integer	average
NICE	Nice value	Integer	average
PCY	Background/Foreground Info	Binary	or
PC	Status of processes	Binary	or
UID	Whether the app is system app	Bool	none

Then, we combine runtime info from different processes of an app. As mentioned in Section 5.1, we collect the apps' runtime information by ps command which provides information about each running process, while *app* is the analysis target. Hence, we combine the runtime information of different processes from the same app. Table 2 shows the features we collected for each app and how they are combined from different processes. We use the difference of runtime information before and after the event as the feature vectors. Here is an example of extracting feature f_1 for an app. App a has two processes: p_1 and p_2. Event e is observed at time t. The process sampling provides the nearest process info logs at time t_1 and t_2, while $t_1 \leqslant t \leqslant t_2$. The f_1 value of p_1 is v_1 at t_1 and v_2 at t_2. The f_1 value of p_2 is u_1 at t_1 and u_2 at t_2. Hence, the processed feature f_1 of a for e is:

$$f_1 a = AVG(v2 - v1,\ u2 - u1)$$

Lastly, we identify the app that triggers it from the users' responses and label all running apps. For each event, we label "1" if an app is selected for it and label "0" if it is not selected.

5.4 Modelling and Precision

The machine learning technique we use for identifying the app is learning to rank [8]. Our scenario is a ranking problem, where we need to select an app that has the highest possibility of invoking the event among a list of running apps. We use RankLib [10], a library that contains several popular ranking algorithms, for the modeling and testing. The model built by RankLib is generic for all apps and all events.

We randomly pick two thirds of the data samples as the training data and the rest as the testing data. We tried all of the eight

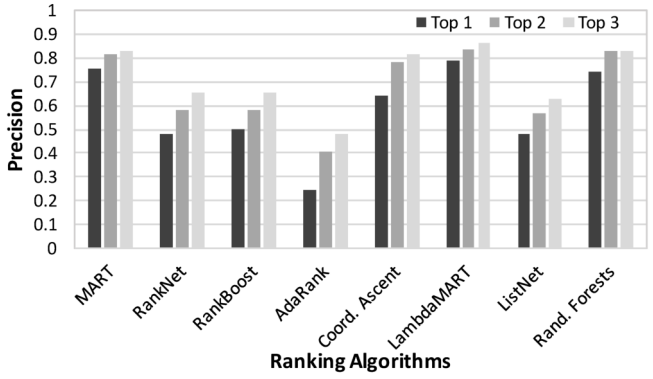

Figure 4: The performance of different ranking algorithms in RankLib library.

algorithms in RankLib with different configurations and compared their performance. Figure 4 presents the precision, the percentage of events the initiator of which can be successfully identified, of the models built by different ranking algorithms. With the LambdaMART algorithm, the precision of UpDroid can reach 80%, and the false alarm (the situation where the app ranked first does not cause the event) rate is around 20%. From our observation, in one data sample, different apps may have the same ranking score, and this brings in a lot of false alarms. Hence, we list the apps with top1, top2 or top 3 ranking scores to see whether they contain the one selected by the inspectors. The precision comparison of the eight algorithms is presented in Figure 4.

6 COMPARISON WITH API HOOKING

Various tools [21, 29, 39] analyze sensitive APIs to reveal the underlying behavior of the target apps. For example, `sendTextMessage()` reveals the behavior of sending SMS. These tools log the APIs called by an app and record their parameters and return values by hooking all sensitive APIs, which typically requires root permission or modification to Android internals. In this section, we present the comparison between UpDroid and the API hooking method on capturing the sensitive behavior.

6.1 Current State of API hooking

To analyze sensitive behaviors through API hooking, we need a list of sensitive APIs and the permissions they require to define the behavior. Sensitive API refers to the API protected by certain permission. Since 2011, researchers have studied to extract the list of sensitive APIs from Android source code [2, 3, 6, 13]. From existing works, static analysis, including code analysis and annotation analysis, is believed to be the most efficient and effective method. However, from our investigation, none of the current methods can provide a complete list of the sensitive APIs. The popular tool Axplorer [3] provides an accurate list of the Android APIs in the Android framework source code, but it misses the analysis of Java APIs and the APIs whose permission checking is in native code. Some popular sensitive APIs, such as `android.hardware.camera2.CameraManager.openCamera()`, are not in the list. DPSPEC [6]

analyzes the annotation of Android source code to identify sensitive APIs. However, it only focuses on the APIs protected by dangerous permissions and needs manual identification to obtain the list. Another work, android-a2p [13], is also based on annotation analysis. It is released on GitHub and is accurate but not complete. In order to cover more APIs for capturing a complete list of sensitive behaviors, we combine the lists from these three works in the comparison.

API hooking is also effective in capturing accessing to sensitive content providers and passing sensitive Intent. DPSPEC and Pscout [2] list the content providers and intents which need dangerous permissions. We also include these sensitive components in our comparison.

In this work, we use an open source tool named EagleEye [23] which is built on the Xposed framework [35] to hook the sensitive APIs and the APIs for accessing content providers and sending intents on a rooted device.

6.2 Permission Coverage Comparison

To see how far UpDroid can go in covering different categories of events, we analyze the permission coverage of UpDroid and that of API hooking (including hooking sensitive APIs, content providers, and Intents). We use 26 *dangerous* permissions and 44 *normal* permissions crawled from Google's official documentation for the comparison. If any sensitive API in the list uses certain permission, we consider that API hooking covers this permission. Also, if UpDroid can capture one kind of behavior which is protected by a permission, we consider that UpDroid covers this permission. This comparison may have inaccuracy since neither sensitive API nor UpDroid cover all the cases that a permission is used. However, this analysis still gives us a hint about what kind of behavior UpDroid and API hooking can capture.

The comparison of dangerous permission coverage is presented in Table 3. The list of sensitive API/Content Provider/Intent we obtain from previous works covers 21/26 *dangerous* permissions, while UpDroid covers 15/26. And 14 permissions can be covered by both methods. As presented in Table 3, most of the ones that cannot be captured by UpDroid are about reading data, e.g., `READ_CALENDAR` and `READ_PHONE_NUMBERS`. This is because UpDroid focuses on the behavior that changes the state of resources on the device while reading normally does not cause any change to them. The comparison of normal permission coverage can be found in Table 5 from Appendix.

6.3 Event Details Comparison

API hooking can provide details about each event or behavior based on the parameters and return value of an API, while UpDroid has a different method of providing detailed information for each event. We present the difference of monitoring each category of events as follows.

Content Provider: While observing content providers, the events come from the changes to the providers. The detailed information of the events can be obtained from changes to the rows in the provider's table. In the case of SMS activities, API hooking logs the `sendTextMessage()` API in apps, while UpDroid observes the `content://sms` content provider. Table 4 shows the information we can obtain from API hooking and content observing. Hooking APIs can get more low-level information, such as the Intent for sending this SMS. UpDroid can get more general information, such as when the SMS request is generated and when the SMS is successfully sent out.

Interrupt: The monitoring based on interrupt sampling observes events from the changing of the number of the corresponding interrupt. It tells whether an event occurs and when it occurs through the changing pattern. Take Bluetooth interrupt as the example. Tracing API `android.bluetooth.BluetoothAdapter.enable()` can detect turning on of the Bluetooth on the devices. On the other hand, UpDroid observes it through recognizing a steep increase of the interrupt. Using Bluetooth continuously (e.g., sharing files) will be represented by a continuous slow increase.

External Storage: As presented in Section 4, the information we can log from the file observers contains the file operation and the

Table 3: The comparison of dangerous permission coverage between API hooking and UpDroid. In this table, ✗stands for none of the permissions in this categorize is covered and ✓stands for all are covered.

Permission	API Hooking			UpDroid
	Sensitive API	Content Provider	Intent	
ACCESS_COARSE_LOCATION	✓	✗	✓	✓
ACCESS_FINE_LOCATION	✓	✗	✓	✓
ADD_VOICEMAIL	✗	✓	✗	✓
ANSWER_PHONE_CALLS	✗	✗	✗	✓
BODY_SENSORS	✗	✗	✗	✗
CALL_PHONE	✓	✗	✓	✓
CAMERA	✓	✗	✓	✓
GET_ACCOUNTS	✓	✗	✗	✗
PROCESS_OUTGOING_CALLS	✗	✗	✓	✓
READ_CALENDAR	✗	✓	✗	✗
READ_CALL_LOG	✗	✓	✗	✗
READ_CONTACTS	✓	✓	✗	✗
READ_EXTERNAL_STORAGE	✓	✓	✗	✓
READ_PHONE_NUMBERS	✗	✗	✗	✗
READ_PHONE_STATE	✓	✗	✓	✗
READ_SMS	✓	✓	✗	✗
RECEIVE_MMS	✓	✗	✓	✓
RECEIVE_SMS	✗	✓	✓	✓
RECEIVE_WAP_PUSH	✗	✗	✗	✗
RECORD_AUDIO	✗	✗	✗	✗
SEND_SMS	✓	✓	✓	✓
USE_SIP	✓	✗	✗	✗
WRITE_CALENDAR	✗	✓	✗	✓
WRITE_CALL_LOG	✗	✓	✗	✓
WRITE_CONTACTS	✗	✓	✓	✓
WRITE_EXTERNAL_STORAGE	✓	✓	✗	✓
Total ✓	21/26			15/26

Table 4: The SMS event details provided by API Hooking and the content observer of UpDroid

Info	API	Content Provider
Destination Address	✓	✓
Source Address	✓	✓
Message Text	✓	✓
Sent Intent	✓	✗
Delivery Intent	✓	✗
Date Initiate	✓	✓
Date Sent	✗	✓
Person	✗	✓

path of the file in external storage. To log the changes to the files, we need to back up the target files and make a comparison. To decrease the overhead, we choose only to record the operations and the paths. Existing API hooking method can hook file operation APIs, such as `java.io.writer.write(String s)`. It tells not only which operation is performed, but also the related content, e.g., the content that is written to a file.

Network: In the case of network activities, UpDroid provides lower level information than API hooking. UpDroid can monitor the low-level activities, e.g., a TCP packet is sent by UID `10080` from `10.0.8.1:38175` to a server at `74.125.24.95:443`. No higher level information, e.g., whether the packet is sent for loading a webpage, will be provided. The parameters and the type of the API imply the behavior of the app and the detailed information related to the behavior. It allows identifying different operations from the called API, e.g., `android.webkit.WebView.load(String URL)` represents loading a URL to a WebView.

6.4 Behavior Outcome Comparison

API hooking logs each attempt at using an API and needs further analysis to find out whether the called API is successfully invoked or not. Even with further analysis, it still misses the results of some app behavior. Contrarily, the four types of events reported by UpDroid represent the behavior that had successfully been performed. This is because it monitors the changes to public resources that will be manipulated by the apps' behavior. Here we compare the differences between UpDroid and API hooking in revealing the outcome of an attempt at performing a certain operation.

API hooking can use several ways to determine whether an API call is successfully called. The first and most apparent one is to check the return value. For example, `android.Bluetooth.BluetoothAdapter.enable()` returns boolean -"true to indicate adapter startup has begun, or false on immediate error". The second way is to check the exceptions thrown by the API. For example, if `sendTexMessage` throws `IllegalArgumentException`, the message is not successfully sent because of empty destination address or text. Another more complicated method is to hook the callbacks as stated in the parameters. For example, `android.hardware.camera2.CameraManager.openCamera(String cameraId, CameraDevice.StateCallback callback, Handler handler)` has a parameter named callback. The callback will be invoked once the camera starts. For some other APIs, the callback is an intent which will be invoked after the API is successfully called. Comparing to the prior two methods, checking whether the API call succeeds or not through the third method needs more advanced API hooking techniques. These techniques should be able to obtain the callback from the API's parameter, hook it and determine whether the callback is invoked due to the API call. Hence, we only consider the first two methods in our analysis of the sensitive APIs.

From our analysis of the sensitive APIs, 154 out of the 400 do not have any implication about the result of the API call. And among the 154, there are 29 which use the permissions that can be covered by UpDroid. Among these 29, there are 14 that UpDroid conveniently reveals the outcome of the attempts. The rest is the APIs that do not change public resources on the devices. For example, `android.net.ConnectivityManager.requestNetwork()` requests network but does not send out packages, so the behavior cannot be detected by UpDroid. UpDroid can determine whether the behavior of the app changes the resources, but it cannot determine which API is used to trigger an event. UpDroid places more emphasis on the result of the app's behavior, while API hooking emphasizes more on the attempt of the app's behavior.

7 CAPABILITY ANALYSIS

In this section, we evaluate the capabilities of UpDroid by analyzing its permission coverage and testing it on several popular apps. We also present the runtime performance of UpDroid evaluated with a popular benchmark app.

7.1 Permission Coverage

To evaluate whether UpDroid can detect the sensitive behavior that requires commonly used permissions, we analyze the permission usage of both malicious and benign apps. We analyze 2000+ malware samples (chosen from 72 malware families) provided by Android Malware Dataset Project [32] and 3000+ apps downloaded from the top chart of Google Play. For each permission, we count the number of apps that declare the permission in the manifest to find out the popular permissions used by malicious and benign apps. The dangerous permission usage is shown in Figure 5. As presented, the permissions from `WRITE_CONTACTS` to `ANSWER_PHONE_CALLS` can be covered by UpDroid. The results show that UpDroid covers the widely used permissions. And it cannot cover the ones for reading private data or the phone states which will not cause any state changing of the observable resources on the device.

7.2 Runtime Experiments

To evaluate how UpDroid performs at runtime for capturing sensitive behavior, we test it on several popular apps, including a communication app WhatsApp, a social networking app Facebook, and an online shopping app Lazada. These apps have more sensitive behavior than most of the malware. We manually run the apps for five minutes while using UpDroid and API hooking to detect the sensitive behavior. We find that UpDroid successfully captures sensitive behavior, such as sending SMS, accessing the camera, opening Bluetooth and so on.

Specifically, we present the experiment on WhatsApp which uses various permissions and compare the results (as shown in Figure 6) of UpDroid and API hooking. The upper figure presents the behavior captured by UpDroid, and the lower one presents the permission usage detected by API Hooking. The upper figure shows that UpDroid detects Bluetooth events, camera events, file operations and network activities of WhatsApp. Compared to API hooking, UpDroid detects the events which manipulate public resources on Android. For network events, UpDroid detects the packages sent out or received, while API hooking reports the access to the network state. Although API hooking can also detect the internet usage, no internet activity is found due to the incompleteness of the sensitive API lists. This also happens on the Bluetooth activities and file operations. As shown in Figure 6, using Bluetooth is monitored by UpDroid at the fifth minutes, but it is not observed by API hooking. UpDroid captures multiple file system operations

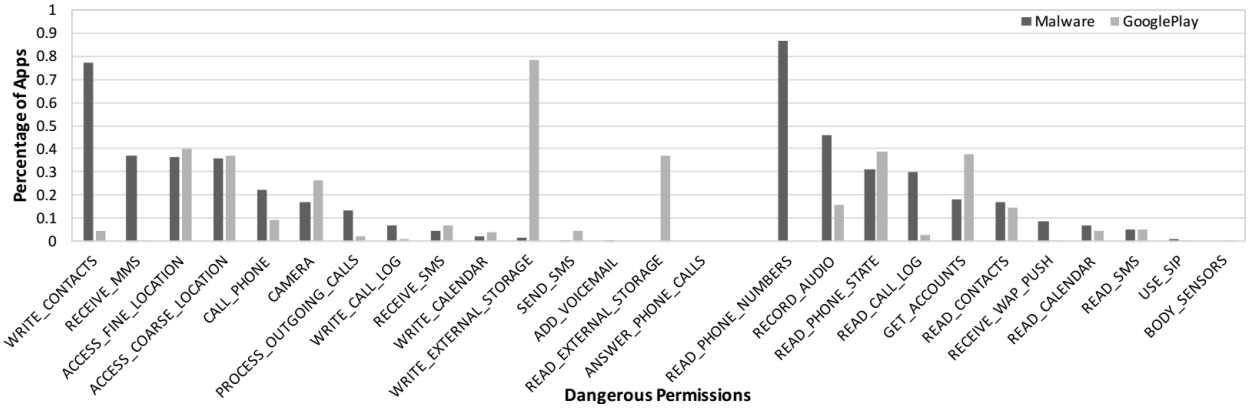

Figure 5: Dangerous permission usage of malware samples from AMD and benign apps from GooglePlay

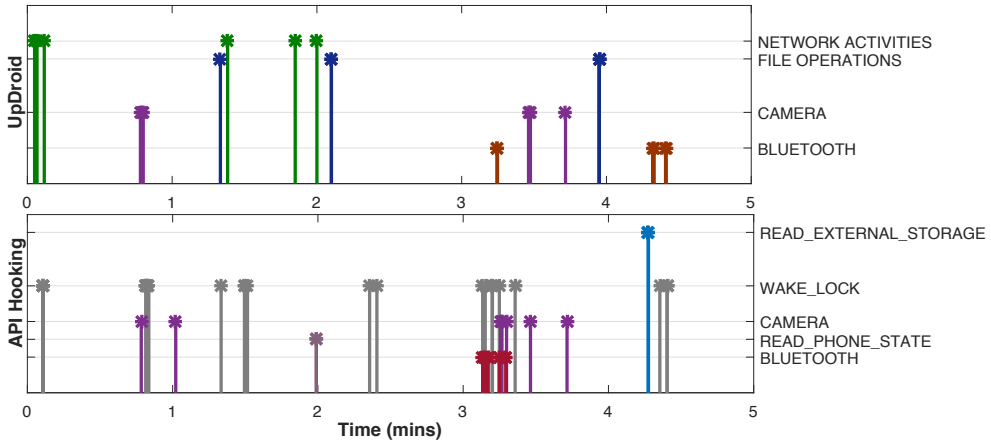

Figure 6: The runtime analysis results of WhatsApp from UpDroid and API hooking

which are not detected by API hooking. It also shows that UpDroid cannot detect the read permissions like READ_PHONE_STATE and WAKE_LOCK.

7.3 Performance

To evaluate the runtime overhead of UpDroid, we run the monitoring module of UpDroid on Nexus 6P with Qualcomm Snapdragon 810 processor and 3GB RAM. We install ten popular apps on the device and keep three of them running in the background. We use one of the most popular benchmarks, Antutu Benchmark, to grade the device with and without UpDroid running on it. The result is presented in Figure 7. The y-axis is the score graded by Antutu. The higher the score is, the faster the CPU runs. In total, UpDroid decreases the benchmark score by 15%. The overhead mainly comes from the high sampling rate of the interrupt numbers and the frequent use of ps command. There is a trade-off between the accuracy and the performance. The evaluation is conducted with a device released in Sep 2015. We believe that the overhead can be decreased on more powerful phones.

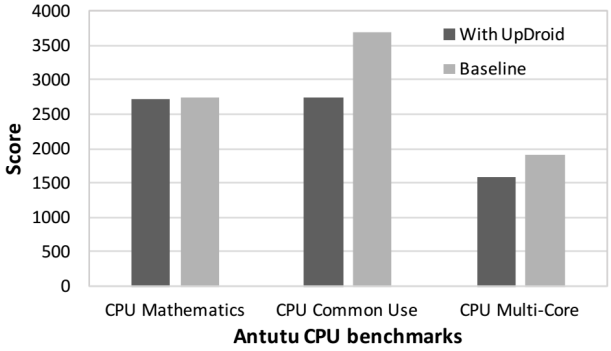

Figure 7: Performance of UpDroid evaluated with Antutu Benchmark

7.4 Discussion

In this section, we discuss how UpDroid can avoid anti-analysis techniques and the possibility of using UpDroid as an attack technique. We also present the limitations of UpDroid.

UpDroid is a dynamic analysis system which is transparent to malware. The monitoring module of UpDroid is implemented with the APIs widely used by app developers (e.g., `ContentObserver` and `VPNService`). The anti-analysis techniques are difficult to evade UpDroid. Among all the APIs, `VPNService` is technically detectable but cannot be used by malware as an indicator, because VPN is widely used by mobile users who need secure and private network. It is also quite popular among Chinese users for accessing blocked websites. Instead of detecting APIs commonly used by normal apps, malware tends to use heuristics which imply the running environment is under analysis (e.g., invalid IMEI number and abnormal GPS info). On the other hand, it would be an advantage when all malware stops its malicious behavior after detecting UpDroid on the users' devices.

UpDroid is designed for analyzing the underlying app behavior, but the techniques used can also be applied to maliciously monitor the users. The monitoring technique that uses `/proc/interrupts` can be applied to side-channel attacks, which monitor sensitive behavior on the device without any permission needed. The app identification model also starts a study to break the process isolation on Android.

The limitation of UpDroid is that it requires access to `/proc` file system which is protected by critical SELinux policies since Android 7. From Android 7, the `ps` command cannot access the process info of other processes. Hence, we need to identify other runtime info, e.g., time for launching the other apps, as the feature of each app in the future. Android 8 prevents third-party apps to access `/proc/interrupts`. Hence, UpDroid may not be significantly effective on devices with Android 8 but still works on a larger proportion of devices with prior Android versions. AppBrain shows that the market share of Android SDK versions prior to 8.0 is 95.4% in April 2018 [1].

8 RELATED WORK

Various dynamic tools/platforms have been proposed for analyzing Android apps underlying behavior. DroidScope [36], CopperDroid [29], VetDroid [39], DroidBox [21] and other tools analyze the API calls, system calls, or other features to reconstruct app behavior. For example, CopperDroid can reconstruct the apps' behavior, e.g., sending SMS, by observing and dissecting the system calls. These tools are based on app instrumentation, framework modification or emulator instrumentation, which need input generator tools to automatically run the target apps. UpDroid can also detect sensitive behavior of Android apps. The difference is that UpDroid can be applied to devices used by the general public while these tools have critical requirements for either the running environment or the target apps.

Andromaly [27], CrowdDroid [7] and other tools can also run on non-rooted devices for app analyzing. These tools detect runtime features, e.g., system call logs and side channel info, of the running apps and use these features to identify whether the app is benign or not with machine learning techniques. UpDroid gathers the running features and uses machine learning techniques to identify the apps that invoke the captured events. Meanwhile, UpDroid generates fine-grained reports of apps, while these tools only classify the apps. App Guardian [38] also gathers side channel info and detects malicious behavior, e.g., on non-rooted devices. However, the detection is based on specific heuristics and only targets runtime information gathering attacks.

BareDroid [24], Ninja [25], Njas [4] and other works provide dynamic analysis techniques which are resistant to anti-analysis techniques. BareDroid is an analysis system which uses a phone cloud for the analysis. It needs to customize the devices in the phone cloud and thus cannot be applied to devices used by the public. Ninja needs to customize the firmware on the Android devices. It is also difficult to be applied to devices used by the public. Njas provides sandboxing for unmodified apps on non-rooted devices. It dynamically loads the target app's APK file to the sandboxing app's context for fully accessing the target app's resources and runtime state. Njas relies on an app database and needs to obtain the APK file of the target app at the same version. Njas cannot sandbox the apps which do not have a readable APK files, e.g., the paid apps. Although these analysis systems are transparent to anti-analysis techniques, they cannot be applied to devices used by the general public directly. Compared to these systems, UpDroid can be easily deployed on the users' devices without any modification to the systems or any requirement on the target apps.

As well as current dynamic analysis techniques, other works also give us inspiration about the runtime monitoring. Diao et al. [12] propose to use the interrupt time series produced by the touchscreen controller to infer the unlock pattern and foreground app. We dig deeper to interrupt to infer the sensitive behavior. MopEye [34] leverages the `VpnService` API to monitor the network usage of the apps. UpDroid also uses this technique to detect network activities on the devices.

9 CONCLUSION AND FUTURE WORK

In conclusion, we present our efforts on the dynamic analysis of app behavior under unmodified and non-rooted devices. We propose UpDroid - a system for dynamically monitoring Android apps' sensitive behavior. It uses different APIs to monitor Android system at runtime and leverages learning to rank technique to identify the initiator of the detected behavior. We use the permission coverage, the runtime experiments and the comparison with the traditional API hooking method to demonstrate the capabilities of UpDroid. The results show that UpDroid can detect sensitive behavior that manipulates the resources of the devices and identify the apps that trigger the behavior. Currently, the number of participants in the *initiator study* is small. In the future, we will obtain labeled data from more users to build a more precise identification model and deploy the system to crowdsourcing for real applications, e.g., malware detection.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Adwait Nadkarni for their valuable comments and suggestions that have helped improve our paper.

REFERENCES

[1] AppBrain. 2018. Top Android SDK versions. (2018). Retrieved April, 2018 from https://www.appbrain.com/stats/top-android-sdk-versions

[2] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. ACM, 217–228.

[3] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of 25th USENIX Security Symposium (USENIX Security '16)*. USENIX Association, 1101–1118.

[4] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '15)*. ACM, 27–38.

[5] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. 2010. An android application sandbox system for suspicious software detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE '10)*. IEEE, 55–62.

[6] Denis Bogdanas. 2017. DPerm: Assisting the Migration of Android Apps to Runtime Permissions. *arXiv preprint arXiv:1706.05042* (2017).

[7] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '11)*. ACM, 15–26.

[8] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning (ICML '07)*. ACM, 129–136.

[9] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE, 429–440.

[10] V Dang. 2013. RankLib. (2013). http://sourceforge.net/p/lemur/wiki/RankLib

[11] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. 2016. DroidScribe: Classifying Android Malware Based on Runtime Behavior. *Mobile Security Technologies (MoST 2016)* 7148 (2016), 1–12.

[12] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P '16)*. IEEE, 414–432.

[13] fgwei. 2017. Android API to Permission Mapping Extractor. (2017). https://github.com/fgwei/android-a2p

[14] Google. 2017. Application security. https://source.android.com/security/. (2017).

[15] Google. 2017. FileObserver. https://developer.android.com/reference/. (2017).

[16] Google. 2017. UI/Application Exerciser Monkey. (2017). https://developer.android.com/studio/test/monkey.html

[17] Heqing Huang, Kai Chen, Chuangang Ren, Peng Liu, Sencun Zhu, and Dinghao Wu. 2015. Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. In *Proceedings of the 10th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS '15)*. ACM, 7–18.

[18] International Data Corporation (IDC). 2017. Smartphone OS Market Share, 2017 Q1. (2017). https://www.idc.com/promo/smartphone-market-share/os

[19] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: automatically generating heuristics to detect Android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, 216–225.

[20] Mohammad Karami, Mohamed Elsabagh, Parnian Najafiborazjani, and Angelos Stavrou. 2013. Behavioral analysis of android applications using automated instrumentation. In *Proceedings of the 2013 IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C '13)*. IEEE, 182–187.

[21] P Lantz, A Desnos, and K Yang. 2017. DroidBox: Android application sandbox. (2017). Retrieved Nov, 2017 from https://github.com/pjlantz/droidbox

[22] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, 224–234.

[23] MindMac. 2016. Android EagleEye. https://github.com/MindMac/AndroidEagleEye. (2016).

[24] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. 2015. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31th Annual Computer Security Applications Conference (ACSAC '15)*. ACM, 71–80.

[25] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards transparent tracing and debugging on arm. In *Proceedings of 26th USENIX Security Symposium (USENIX Security '17)*. USENIX Association, 33–49.

[26] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the 7th European Workshop on System Security (EuroSec '14)*. ACM, Article 5, 5:1–5:6 pages.

[27] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems* 38, 1 (2012), 161–190.

[28] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, 1808–1815.

[29] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS '15)*.

[30] Timothy Vidas and Nicolas Christin. 2014. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS '14)*. ACM, 447–458.

[31] Xiaolei Wang, Sencun Zhu, Dehua Zhou, and Yuexiang Yang. 2017. Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*. ACM, 350–361.

[32] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*. Springer, 252–276.

[33] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS '16)*.

[34] Daoyuan Wu, Rocky K. C. Chang, Weichao Li, Eric K. T. Cheng, and Debin Gao. 2017. MopEye: Opportunistic Monitoring of Per-app Mobile Network Performance. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 445–457. https://www.usenix.org/conference/atc17/technical-sessions/presentation/wu

[35] Xposed. 2017. Welcome to the Xposed Module Repository! http://repo.xposed.info/. (2017).

[36] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.. In *Proceedings of 21th USENIX Security Symposium (USENIX Security '12)*. USENIX Association, 569–584.

[37] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. 2014. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Proceedings of the 2014 European symposium on research in computer security (ESORICS '14)*. Springer, 163–182.

[38] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and XiaoFeng Wang. 2015. Leave me alone: App-level protection against runtime information gathering on android. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P '15)*. IEEE, 915–930.

[39] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. 2013. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13)*. ACM, 611–622.

[40] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. 2013. AppInk: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS '13)*. ACM, 1–12.

[41] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P '12)*. IEEE, 95–109.

APPENDIX

Table 5 presents the comparison of normal permission coverage between API hooking and UpDroid. From this table, API hooking covers 24/44 normal permissions, while UpDroid covers only 5/44. Although UpDroid does not cover many normal permissions, it covers the popular ones, such as `BLUETOOTH` and `NFC`.

Table 5: The comparison of normal permission coverage between API hooking and UpDroid. In this table, ✗stands for none of the permissions in this categorize is covered and ✓stands for all are covered.

Permission	API Hooking			UpDroid
	Sensitive API	Content Provider	Intent	
ACCESS_LOCATION_EXTRA_COMMANDS	✓	✗	✗	✓
ACCESS_NETWORK_STATE	✓	✗	✗	✗
ACCESS_NOTIFICATION_POLICY	✗	✗	✗	✗
ACCESS_WIFI_STATE	✓	✗	✗	✗
BLUETOOTH	✓	✗	✓	✓
BLUETOOTH_ADMIN	✓	✗	✓	✓
BROADCAST_STICKY	✓	✗	✗	✗
CHANGE_NETWORK_STATE	✓	✗	✗	✗
CHANGE_WIFI_MULTICAST_STATE	✗	✗	✗	✗
CHANGE_WIFI_STATE	✗	✗	✗	✗
DISABLE_KEYGUARD	✓	✗	✗	✗
EXPAND_STATUS_BAR	✗	✗	✗	✗
GET_PACKAGE_SIZE	✓	✗	✗	✗
INSTALL_SHORTCUT	✗	✗	✓	✗
INTERNET	✓	✓	✗	✓
KILL_BACKGROUND_PROCESSES	✓	✗	✗	✗
MANAGE_OWN_CALLS	✗	✗	✗	✗
MODIFY_AUDIO_SETTINGS	✓	✗	✗	✗
NFC	✓	✗	✓	✓
READ_SYNC_SETTINGS	✓	✓	✗	✗
READ_SYNC_STATS	✓	✗	✗	✗
RECEIVE_BOOT_COMPLETED	✓	✗	✓	✗
REORDER_TASKS	✓	✗	✗	✗
REQUEST_COMPANION_RUN_IN_BACKGROUND	✗	✗	✗	✗
REQUEST_COMPANION_USE_DATA_IN_BACKGROUND	✗	✗	✗	✗
REQUEST_DELETE_PACKAGES	✗	✗	✗	✗
SET_ALARM	✗	✗	✓	✗
SET_WALLPAPER	✓	✗	✗	✗
SET_WALLPAPER_HINTS	✓	✗	✗	✗
SIGNAL_PERSISTENT_PROCESSES	✗	✗	✗	✗
TRANSMIT_IR	✓	✗	✗	✗
USE_FINGERPRINT	✓	✗	✗	✗
VIBRATE	✓	✗	✗	✗
WAKE_LOCK	✓	✗	✗	✗
WRITE_SYNC_SETTINGS	✓	✗	✗	✗
Total ✓	26/35			5/26