Automated Software Engineering (2023) 30:10
https://doi.org/10.1007/s10515-023-00378-w

DroidHook: a novel API-hook based Android malware dynamic analysis sandbox

Yuning Cui[1,2] · Yi Sun[1,2] · Zhaowen Lin[1,2]

Received: 10 October 2022 / Accepted: 11 February 2023 / Published online: 24 February 2023

Abstract

With the popularity of Android devices, mobile apps are prevalent in our daily life, making them a target for attackers to steal private data and push advertisements. Dynamic analysis is an effective approach to detect runtime behavior of Android malware and can reduce the impact of code obfuscation. However, some dynamic sandboxes commonly used by researchers are usually based on emulators with older versions of Android, for example, the state-of-the-art sandbox, DroidBox. These sandboxes are vulnerable to evasion attacks and may not work with the latest apps. In this paper, we propose a prototype framework, DroidHook, as a novel automated sandbox for Android malware dynamic analysis. Unlike most existing tools, DroidHook has two obvious advantages. Firstly, the set of APIs to be monitored by DroidHook can be easily modified, so that DroidHook is ideally suitable for diverse situations, including the detection of a specific family of malware and unknown malware. Secondly, DroidHook does not depend on a specific Android OS but only on Xposed, so it can work with multiple Android versions and can perform normally on both emulators and real devices. Experiments show that DroidHook can provide more fine-grained and precise results than DroidBox. Moreover, with the support for real devices and new versions of Android, DroidHook can run most samples properly and acquire stronger detection results, compared to emulator-based tools.

Keywords Android · Mobile malware · Dynamic analysis · Sandbox

✉ Yi Sun
sybupt@bupt.edu.cn

Yuning Cui
cuiyn@bupt.edu.cn

Zhaowen Lin
linzw@bupt.edu.cn

[1] School of Computer Science (National Pilot Software Engineering School), Beijing University of Posts and Telecommunications, Beijing, China

[2] National Engineering Laboratory for Mobile Network Security (No. [2013]2685), Beijing, China

1 Introduction

Android is the leading mobile operating system with about 71.47% market share in August 2022 (Statista 2022), attracting both developers and attackers. Android has been the target of 97% mobile malware (Kelly 2014), and there are 2.48 million malware and 2.97 million potentially unwanted apps aiming at Android in 2020 (AV-TEST 2020). The increasing growth of the Android market has put forward urgent requirements for malware detection. What is worse, attackers are now using various evasion strategies including obfuscation, reflection APIs, and anti-emulator techniques, which poses a huge challenge for detectors.

As a consequence, researchers have proposed a wide range of approaches for Android malware detection. They can be broadly divided into two categories: *static analysis* and *dynamic analysis*. For the former, samples are disassembled as source code, manifest file, and resource files. Then, static features such as sensitive API calls, opcode, and permission requests are extracted. Approaches based on static analysis can be quick and large-scale, yet they are vulnerable to evasion attacks, for example, code obfuscation, encryption, and API reflection. Besides, some apps do require sensitive permissions or need to use sensitive APIs. Therefore, the existence of sensitive permissions and APIs does not always lead to attacks (Cai et al. 2018). These shortcomings make static analysis methods not always the best choice.

As for dynamic analysis, samples are executed on an instrumented sandbox environment to extract runtime features, such as system interaction and network access. Dynamic approaches can be more robust when facing static obfuscation, however, since the monitoring environment currently in use is generally a virtual Android device, dynamic analysis methods may be severely affected by anti-emulator techniques. Moreover, to instrument an environment usually needs to modify the source code of samples or Android OS, and this affects the compatibility of dynamic analysis.

DroidBox[1] is the first and the state-of-the-art open-source dynamic analysis sandbox, and it has been used by many dynamic analysis approaches (Gajrani et al. 2020; Sugunan et al. 2018; Feng et al. 2018; Alzaylaee et al. 2016; Chang et al. 2016; Lindorfer et al. 2014). DroidBox provides a modified Android virtual device (AVD) file for emulator and several shell scripts to make itself work right out of the box. However, DroidBox has the following three disadvantages. Firstly, DroidBox is seriously out of date. The Android version of DroidBox's virtual device is 4.1.1, which was released in July 2012 and less than 0.1% of devices are using this or even older version.[2] During experiments, we found that at least half of the sample apps no longer support this version. Secondly, DroidBox is based on a 32-bit ARM emulator. However, desktop computers and servers generally use x86 or x86_64 architecture. The architecture of today's Android devices has also been upgraded to 64-bit ARM, so choosing 32-bit ARM as the architecture of monitoring environment is not necessary. As a result, the emulator is tend to run slower and execute fewer program

[1] https://github.com/pjlantz/droidbox.

[2] https://developer.android.com/about/dashboards.

instructions within the same time. Moreover, for malware detection, emulators have been shown to be less effective than the real devices (Alzaylaee et al. 2017). Finally, DroidBox has a few flaws. The most fatal one is, since DroidBox is based on TaintDroid, it only supports network tracking of tainted data, which we believe is far from sufficient. For example, it is common for a malicious app to access a series of APIs, not only for acquiring necessary information, but also for commands from attacker's C &C server. DroidBox's report of a network request has only IP address and port, while in many cases what's really valuable are domain names, URLs, and delivered content. All of the above makes it difficult for DroidBox to always get accurate and useful detection results.

In this paper, we propose a prototype system, *DroidHook*, as a sandbox for Android malware dynamic analysis. DroidHook can run most of the samples automatically and gather the detection reports in JSON format, which can be further analyzed by subsequent methods such as machine learning models. DroidHook is based on Xposed,[3] which allows developing modules to change the behavior of Android OS and apps without modifying any source code. DroidHook is not dependent on a particular system – any real devices or emulators supported by Xposed are suitable for DroidHook. Furthermore, DroidHook is designed to be flexible and the set of monitored APIs can be easily expanded for different scenarios. We totally collect 3,389 benign samples and 8,314 malicious samples for evaluation. Experiments show that DroidHook can support more recent samples for both emulator and real devices, and can provide more fine-grained and precise results than DroidBox. We release DroidHook as an open-source tool and make the SHA-1 hash values of the samples publicly available on GitHub.[4]

In summary, the contributions of this work are listed below.

- We research the behavior of common Android malware families and use it as a basis to propose five categories of relevant APIs to be monitored for our sandbox.
- We design DroidHook, which is a lightweight, compatible, and easily expandable tool, as an automated sandbox for Android malware dynamic analysis. DroidHook supports sensitive APIs configuration when monitoring sample behaviors and the sandbox can be used to test real-world devices. We release DroidHook as an open-source tool, which could be useful for the community to further analyze the new samples in the future.
- We perform experiments on DroidHook with Google Play's most popular apps and AndroZoo, the authoritative Android malware database, as the source of benign and malicious samples. Evaluations show that DroidHook can handle recent samples and observe more behavior of the samples based on real devices, compared to the state-of-the-art sandbox, DroidBox.

[3] https://repo.xposed.info.
[4] https://github.com/DroidHook/.

The structure of the paper is as follows. In Sect. 2 we summarize the related work. In Sect. 3 we make a brief introduction to the background of this work. In Sect. 4 we describe the design of the DroidHook prototype system. In Sect. 5 we collect 3,389 benign samples and 8,314 malicious samples as two datasets to evaluate DroidHook and make comparisons with DroidBox.

2 Related work

In this section, we summarize and introduce the typical Android malware detection methods.

2.1 Static detection

Previous Android malware static detection methods are mainly based on features and machine learning classifiers. Peiravian and Zhu (2013) combined permission and API calls as features, which are statically extracted from packed app files, and use machine learning methods to detect malicious Android apps. Aafer et al. (2013) proposed DroidAPIMiner, a robust and lightweight Android malware classifier based on features captured at the API level. Arp et al. (2017) proposed DREBIN, which performed a broad static analysis and get as many features as possible, including permissions, suspicious API calls, app components, intents, etc.

Recent studies have focused on using more advanced methods of feature extraction and analysis. Fan et al. (2018) constructed frequent subgraphs to represent the common behaviors of malware samples that belong to the same family. Onwuzurike et al. (2019) proposed MaMaDroid, which built a behavioral model in the form of a Markov chain, from the sequence of abstracted API calls. Xiao et al. (2019) considered a system call sequence as a sentence and built a classifier based on LSTM model. Gao et al. (2021) mapped apps and Android APIs into a large heterogeneous graph based on graph convolutional network, solving problems from the perspective of a node classification task.

Some studies focus on a particular type of malware. One of the more notable types is repackaging, which allows attackers to inject malicious code into popular apps. Fan et al. (2017) proposed DAPASA, which generated sensitive subgraphs to profile the most suspicious behavior of an app and extract features for piggybacked malware detection. Similarly, Tian et al. (2017) propose a new Android repackaged malware detection technique based on code heterogeneity analysis. Chen et al. (2019) introduced a new attacking method that generated adversarial Android malware samples by using the repackaged approach. Nicheporuk et al. (2020) used API calls and permissions as features to train classifiers based on convolutional neural network.

In recent years, several research results have proposed program analysis approaches to resolve sophisticated programming languages. In response to malicious developers' usage of reflection to hide malicious operations, Sun et al. (2021) propose DroidRA, which reduces reflection calls to a composite constant

propagation problem and infers the value of the reflection target, then replaces the reflection call with the corresponding Java call. Samhi et al. (2022) propose JuCify, which extracts and merges call graphs of native code and bytecode, allowing static analyzers to uncover situations where malware relies on native code to hide calls to other sensitive code.

2.2 Dynamic detection

How to achieve the instrumentation of the test environment is the key to dynamic detection approaches. Since Android is an open-source project, one of the commonly used methods is to modify the system's code or runtime files. Enck et al. (2014) proposed TaintDroid, which could support taint tracking on different levels and provide real-time analysis by leveraging Android's virtualized execution environment, Dalvik. In addition, DroidBox is based on TaintDroid and also achieves its effect by modifying Dalvik, the Android system runtime. Sun et al. (2016) proposed TaintART, which implemented taint analysis on the newer ART environment and achieved a high performance improvement. Cho et al. (2018) proposed DexMonitor, which analyzed samples by placing hooks in the Dalvik virtual machine at the point where a Dalvik instruction is about to be executed. Xue et al. (2018) proposed NDroid, which implemented taint tracking and analysis of both Java code and native code by modifying Dalvik. Cai et al. (2019) proposed DroidCat, which is an advanced method for classifying Android apps with high robustness and the ability to handle reflection-based evasion attacks.

Another way is to leverage injection techniques provided by Linux since Android uses a modified Linux kernel. Zheng et al. (2014) proposed DroidTrace, which is a ptrace based dynamic analysis system to monitor selected system calls and classify the payload behaviors through system call sequence. Xu et al. (2016) used strace, a Linux utility to collect system call invocations, and produced a feature vector of each app.

In addition, there is a method to monitor system call information by modifying the simulation environment. Tam et al. (2015) implemented monitoring of app behavior by running the original Android image in a modified QEMU emulator. Based on this research, a variety of dynamic detection methods are proposed. For example, Sihag et al. (2021) proposed an obfuscation-resilient approach and achieved improved detection results by collecting behavioral features of apps running in virtual environments.

2.3 Hybrid detection

Hybrid approaches combine the advantages of static and dynamic methods to improve detection efficiency and accuracy. There have been only a few studies utilizing hybrid detection. Arshad et al. (2018) proposed SAMADroid, which is a 3-level hybrid malware detection model to combine the benefits of static/dynamic analysis, local/remote host, and machine learning techniques. Wang et al. (2019) proposed a hybrid model based on deep autoencoder and convolutional neural network. Gajrani

et al. (2020) proposed EspyDroid+ for reflection analysis, which pruned apps using static methods for more efficient dynamic detection.

3 Background

In this section, we provide a brief introduction to Android and Xposed framework.

3.1 Android apps

Android apps are generally written in JVM languages and native C/C++, then run over process virtual machines like Dalvik or ART. Starting with Android 4.4, ART (Android Runtime) has been introduced as a new runtime environment. ART has better performance, more security and debug features than the original Dalvik. In 2017, the Android team announced that they would provide first-class support for Kotlin. Since then, Kotlin has been getting attention from developers, and now more than 60% of professional Android developers are using Kotlin according to the official website.

There are four main Android app components: activities, services, content providers, and broadcast receivers. An activity usually contains some user-operable GUI interfaces that display content and handle user input. A service is a background process. A content provider is used to manage data, either privately or shared with other apps. A broadcast receiver is used for communication between Android OS and apps.

Android apps are usually distributed as Android Package (APK) files. An Android APK typically contains the following files: *manifest file*, describing the package name, version, permissions required and main activity; *classes.dex file*, classes complied as DEX file to run on ART; *library files*, complied code which is platform dependent; and *resource files*, usually UI layout files and images.

3.2 Android sensitive APIs

Malware generally relies on sensitive APIs to achieve malicious behavior. There are various APIs to acquire personal data, push advertisements, communicate with a command and control (C &C) server, etc. We consider APIs that obtain or manipulate sensitive data and are protected by permissions as sensitive APIs. For example, `telephony.TelephonyManager.getImei()` is often used in a variety of scenarios to obtain the device's IMEI. It should be noted that this API is also heavily used in benign apps to identify users and count the total number of devices that have the app installed. However, none of these APIs are designed for attackers, and we can not simply classify an app as malicious just because it calls a lot of sensitive APIs.

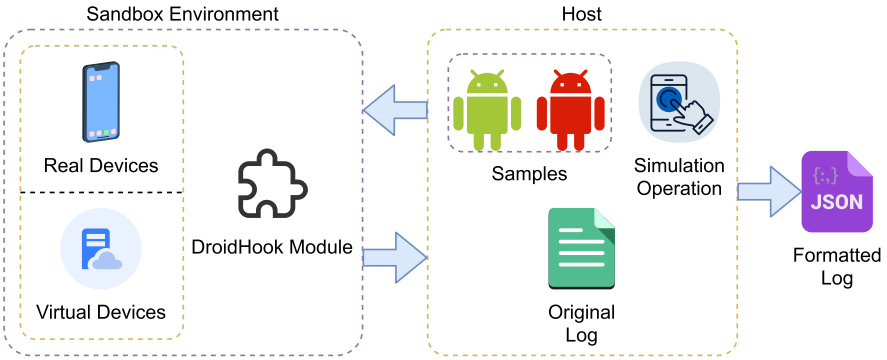


Fig. 1 DroidHook framework

3.3 Xposed framework

Xposed is a framework that allows users to develop modules and make changes to an app without modifying its code. The initial version of Xposed is developed by rovo89. It takes control of all app processes by replacing the Android Zygote process (since Zygote is the parent of all app processes). Currently, Xposed has been followed and maintained by the open-source community. Many community-derived projects can support the latest version of Android OS and are compatible with modules developed for the original Xposed. By leveraging a variety of advanced techniques (for example, app-virtualization), they achieve a higher level of OS compatibility.

Xposed itself does not modify the app behavior, and it requires developers to write modules to do so. DroidHook provides an Xposed module that implements monitoring of the specified APIs.

4 Methodology

In this section, we firstly introduce Root APIs and the approach we apply to determine them. Then, we illustrate the two important components of DroidHook: **sandbox environment** and **host**. Figure 1 shows the whole framework of DroidHook. Sandbox environment provides Android runtime and monitors the Root APIs of given sample via DroidHook module, which can work on both real devices and virtual environments. Host controls the whole detection process, such as feeding samples, simulating user input, acquiring original logs and generating reports.

4.1 Root API

Generally, Android malware relies on APIs to perform attacks. We introduce Root API, which is the most basic API of malicious behavior, for subsequent

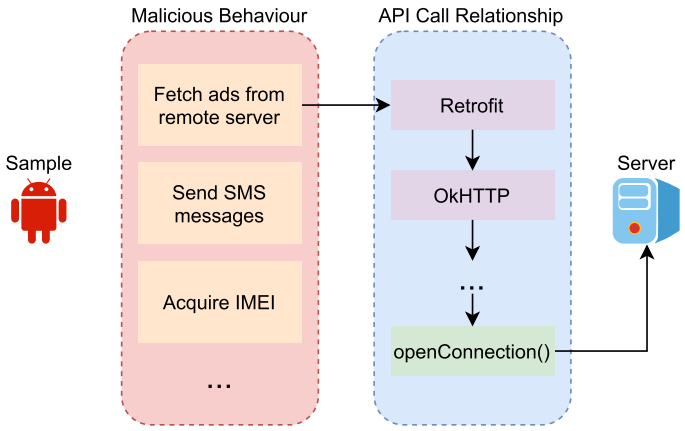


Fig. 2 Network API call relationship of a sample

monitoring and analysis. For example, some samples fetch ads from remote servers and push them to users, and the prerequisite is calling the API for network access. Android developers prefer third-party HTTP clients such as Retrofit, which wraps the low-level APIs provided by Android for efficiency. The Root APIs we would like to pay close attention to are the low-level APIs for Android network requests, rather than the API for HTTP clients. It is not possible to monitor all HTTP clients, but their corresponding low-level APIs are the same. By correctly determining the Root API, DroidHook improves compatibility with a variety of scenarios. Figure 2 shows the API call structure of a sample, and here the Root API is `openConnection()`. Root APIs should be decided by empirical analysis of malware samples and specific application scenarios. For example, when detecting whether a sample is pushing ads, it is necessary to know whether the ad-related URLs are visited, so the Root APIs may include network-related APIs.

For the following evaluations, we firstly summarize the malicious behavior of each malware family, as illustrated in Table 1. Then, we intuitively select a few relevant APIs in each malicious behavior as shown in Table 2 for further comparison. For each type of malicious behavior, we present the reasons for choosing these APIs. We would like to clarify that although we have chosen the above API for evaluation, it would be easy to continue adding other APIs in real-world scenarios.

- *Steal data* For this category of malicious behavior, we have selected four commonly used APIs. Their purposes are easy to see from their names, which are to get device IMEI, phone number, device ID and contact data.
- *Communicate with command and control (C &C) servers* For this category of malicious behavior, we select the Root API `openConnection()` for evaluation. In addition, Zungur et al. (2020) points out that some APIs may communicate directly using sockets, so we also consider the relevant API as Root API.
- *Send SMS* These two APIs can send text and data based on SMS respectively.

Table 1 Typical malicious behaviors of malware families

Family	Malicious Behavior				
	Steal Data	C &C Servers	SMS	Install App	Ads
AdWo	√				√
Airpush					√
BaseBridge		√			
Boqx		√			
Boxer			√		
Clicker					√
Dowgin					√
DroidDreamLight	√	√			
DroidKungfu				√	√
Ewall	√	√			
FakeAngry	√	√			
FakeDoc	√	√			
FakeInst			√	√	
FakePlayer			√		
Geinimi	√	√	√		
GingerMaster	√	√			
GoldDream	√	√			
JSmsHider		√			
Kmin		√	√	√	
Kuguo					√
LoveTrap	√	√	√		
Pjapps	√	√	√		
Plankton	√	√			

Table 2 Root APIs for malicious behaviors

Malicious Behavior	Root API
Steal Data	`telephony.TelephonyManager.getImei()`
	`telephony.TelephonyManager.getLine1Number()`
	`telephony.TelephonyManager.getDeviceId()`
	`android.content.ContentResolver.query()`
C &C Servers	`java.net.URL.openConnection()`
SMS	`telephony.SmsManager.sendTextMessage()`
	`telephony.SmsManager.sendDataMessage()`
Install App	`Intent.setDataAndType()`
Ads	`NotificationManager.notify()`
	`NotificationManagerCompat.notify()`

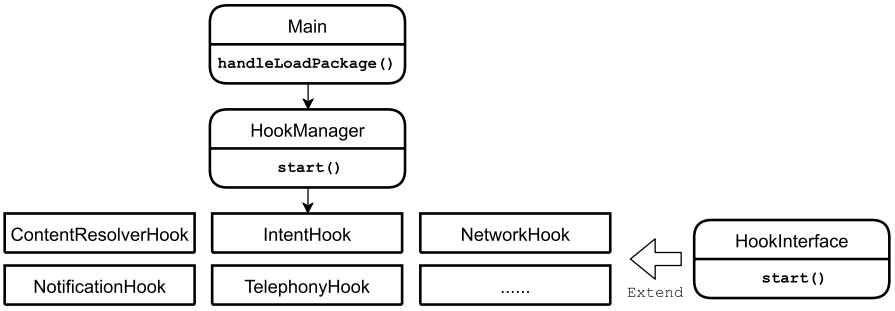


Fig. 3 Structure of DroidHook Xposed module

- *Install app* This category contains two APIs: `android.content.ContentResolver.query()` and `Intent.setDataAndType()`. The former API is chosen because malware usually accesses private information such as contacts and messages by constructing and querying related URIs. For example, if the first parameter of this API is "content://sms/", then the caller is querying messages; if the first parameter is "content://com.android.contacts/contacts", the caller is querying contacts. The latter is because developers need to declare the type of a file as an APK before requesting an installation (without root permission). If the second parameter of this API is "application/vnd.android.package-archive", then the caller is probably about to install the file as an APK.
- *Push advertisements* For this category of malicious behavior, we simply choose the two APIs which can be used for push notifications.

4.2 Sandbox environment

We design and implement an Xposed module for our sandbox environment to hook Root APIs. Figure 3 shows the structure of this module. For the convenience of communication between the sandbox environment and host, we specify the package name of the app to be monitored by modifying a certain file (/sdcard/PackageName), and when the sandbox environment starts, it checks this file and identifies the app. Since the module's code is loaded on every app that the OS runs in the original Xposed, the module firstly determines whether the current app's package name matches the one to detect. If it matches, the module will call the `start()` method of `HookManager`. `HookManager` then calls the list of hooks we have programmed. Moreover, we provide `HookInterface` and a series of utility classes to make creating hookers easier. Each Root API is hooked and their critical parameters are monitored according to Android API reference.[5] Reports are output directly together with Xposed's logs. We can get the entire logs from Xposed

[5] https://developer.android.com/reference.

Manager, which is an app provided by Xposed to manage the whole framework and all modules. Then, logs of our module will be extracted by the host and formatted as a report, which is further discussed in Sect. 4.3.

When developing the module, we pay special attention to its extensibility for the convenience to adapt it to future API changes or porting to other scenarios such as software testing. We make the majority of the code modularized and abstracted so that when changes or additions are necessary, they can be simply modified according to the latest API specification, without having to keep watch on things like log exporting.

We conduct the main work under Android 6 because we also have to consider the compatibility of real device used for experiments. But since we avoid using features related to Android OS or Xposed framework versions, our module works on a variety of versions and devices. For example, our module is available for the more recent Android 11, thanks to the open-source community's continued support of Xposed. Moreover, experiments have pointed out that our module is suitable for both real devices and virtual environments.

4.3 Host

We design a monitor running on the host to cooperate with the sandbox environment and automate the detection of samples. Tasks of the monitor include sample installation, environment preparation, test events generation, and reports extraction. The host does not need to consume excessive computing resources. In our experiments with virtual environments, we deploy the host on the same server as the sandbox environment; for the real device, we use a lightweight ARM-based server, which will be further introduced in Sect. 5.1.2.

Algorithm 1 Monitor

Input: List of apps to be tested App_{list}
Output: List of exported logs Log_{list}
1: **for** each $app \in App_{list}$ **do**
2: setPackageName(app)
3: deviceReboot()
4: installApp(app)
5: grantPrivileges(app)
6: startTesting(app)
7: cleanEnvironment()
8: Log = getLog()
9: Log_{list}.add(Log)
10: **end forreturn** Log_{list}

Algorithm 1 shows the whole workflow of the monitor. Firstly, for each sample in the testing waitlist, the monitor determines its package name and writes it to /sdcard/PackageName. We get the details of the sample using *aapt dump* and

Table 3 Sample datasets

Id	Category	Source	Sample Amount
GP	Benign	Google Play Store	3389
AZ	Malicious	AndroZoo	8314

extract the package name using a regular expression. Secondly, the monitor reboots the device via *adb shell reboot now* command to enable monitoring of the new package name. Thirdly, once the device boots up, the monitor installs the sample and grants all requested privileges in AndroidManifest.xml file. We determine if the device is already booted by getting the status via *getprop sys.boot_completed*, and use *pm grant* to grant privileges. Fourthly, the monitor initiates the testing process, feeds various events into the app, and stops the test after a certain time. We simply choose *Monkey* as the same as DroidBox to generate pseudo-random streams of user events. We can assume that, since the same user event simulation tools is used, DroidHook and DroidBox have very similar code coverage. Finally, the monitor gets test logs and cleans up the environment, including stopping and uninstalling current app and cleaning up old logs so that they will not affect subsequent tests.

5 Experiment and evaluation

In this section, we present the experimental evaluation of DroidHook. Firstly, we introduce our datasets and the experiment environment. Then, we perform an analysis of the datasets. Finally, we compare DroidHook in detail with the current state-of-the-art tool DroidBox, and empirically show that DroidHook outperforms DroidBox in several aspects.

5.1 Prerequisites

5.1.1 Datasets

As listed in Table 3, we totally gathered 2 datasets including 11,703 APK files. These samples were collected from the following two sources.

- *GP* We collected the top 100 most popular apps from 34 categories in Google Play Store, and finally got 3389 samples that can be run properly in our analysis as our benign dataset. The remaining 11 samples were not added to the dataset for two main reasons: firstly, we could not obtain its original APK file; secondly, the samples were designed totally for Wear OS, so they did not run or display properly on the experimental devices. Categories include Art and Design, Books and Reference, Game, Music and Audio, Weather, etc. Samples have been sent to VirusTotal for further examination to ensure that they were all benign.
- *AZ* These samples were downloaded from AndroZoo, based on the following two constraints. Firstly, we only consider samples released after January 1, 2020,

to ensure that they are representative of the latest security threats. Secondly, a sample is considered if it was detected to be malicious/suspicious by five or more anti-virus engines. It is a fairly loose metric, as we wanted to include some potentially unwanted apps and adware that are not "seriously" malicious to get a full picture of DroidHook's performance. Finally, we collected 8314 samples.

We are not able to share the sample files directly without authorization. However, we have published the SHA-1 checksum lists[6] of the two datasets in order to facilitate the reproduction of the following evaluations.

5.1.2 Implementation environment and setup

In our experiments, there are three dynamic analysis sandboxes to be evaluated: DroidBox (**DB**), DroidHook under virtual environment (**DHV**) and DroidHook under real device (**DHR**). The DroidBox we utilize here comes from AndroPyTool (Martín et al. 2019), which integrates many well-known tools and is a handy tool for extracting features from Android APK samples.

We run and evaluate DroidHook's module on both real device and virtual environment. Considering the version support of the real devices used in the experiments, we choose Android 6.0 (API level 23) for DroidHook as the experiment environment for the following two reasons. Firstly, Android 6.0 begin to introduce a new permissions model named Runtime Permissions, where users can now directly manage app permissions at runtime,[7] and we would like to demonstrate the support of DroidHook for this important new security feature. Secondly, Android 6.0 provides better support and compatibility with the samples and real devices chosen for the experiments. The real device is a OnePlus 3 phone with a Snapdragon 820 processor and 6 GB RAM. The phone was firstly used in daily life for a while before the experiments to bring it closer to the actual environment. The virtual machine is built with the *emulator* tool which comes with Android SDK. We allocate 1536 MB RAM for it and simply keep all other settings as default. Since some malware may take advantage of root access to modify the OS or try to stay persistent, we utilize the following methods to protect the environment: firstly, the superuser management tool will not grant root privileges to any app once the experiments begin; secondly, when DroidHook is running on a virtual device, we will replace virtual hard disk files with fresh ones before each monitoring session.

DroidBox and virtual devices with DroidHook module are run on a Ubuntu 16.04 server, which has an Intel Xeon E5-2620 CPU and 32 GB memory. DroidHook host is run on a lightweight Linux server with an ARM Cortex-A53 CPU and 2 GB memory.

[6] https://github.com/DroidHook/checksums.

[7] https://developer.android.com/about/versions/marshmallow/android-6.0-changes.

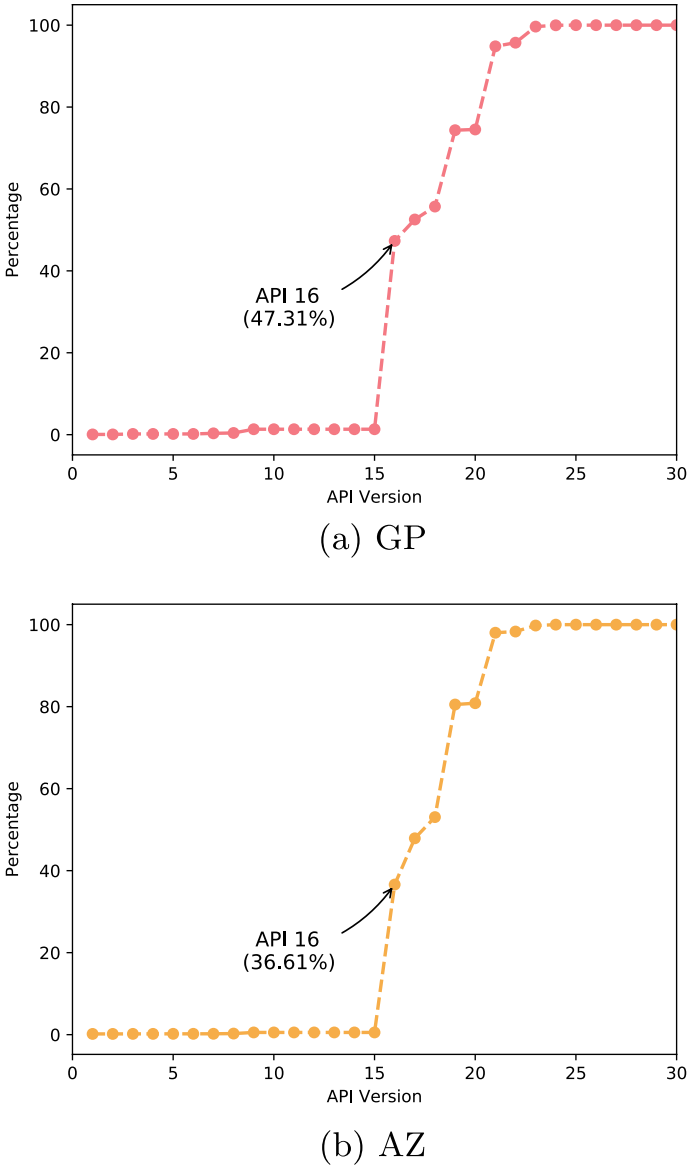
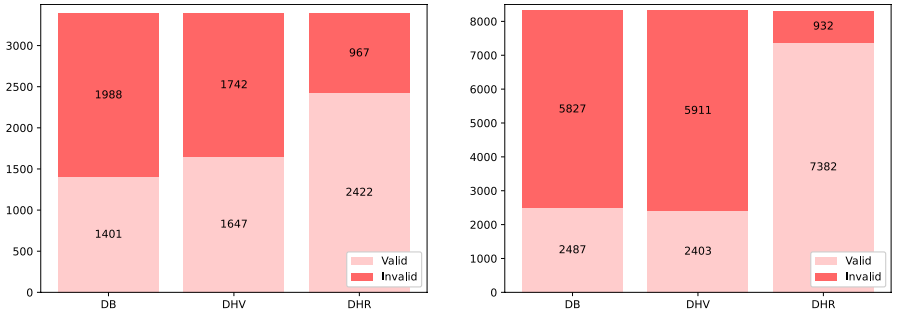


Fig. 4 Percentage of minimum API level required for sample

5.2 Sample minimum supported API level

In this section, we analyze the collected samples and discuss the results. We obtain the minimum supported Android version for each sample and point out that DroidBox with Android API 16 is too old to support the running of today's samples, let alone the monitoring and detection afterwards.



(a) GP (b) AZ

Fig. 5 Analyzability of samples in the two datasets

5.2.1 Minimum supported API level

We calculate the percentage of samples that can be supported by different API levels, as shown in Fig. 4. The minimum supported Android API level is extracted from each sample's manifest file. Devices with the same or higher API level should be able to run the sample normally. The ratios of sample supported by API 16, which is the API level of DroidBox, are marked on the figure. It can be found that DroidBox is too old for most of today's samples, whether benign or malicious.

5.3 Comparison

In this section, we make a comparison between DroidHook and DroidBox in terms of analyzability and quality of reports. In the following experiments, we run each sample in **DB**, **DHV** and **DHR** for the same duration, which ensures the objectivity of these experiments by using roughly the same resources in each environment.

5.3.1 Analyzability

One of the important purposes of dynamic detection is to obtain a detailed report of each sample. Therefore, we firstly consider whether each sample can be supported and output reports under DroidHook and DroidBox, i.e., the analyzability of a sample under these two sandboxes. We evaluate the analyzability of the two sandboxes by the ability to provide detailed reports of the detection for each sample. This evaluation method comes from the following intuition: for malicious samples, in order to achieve their malicious purposes, they generally need to call the APIs provided by the Android, which should be recorded and reported by the sandbox; for benign samples, they generally also need to use the APIs provided by the system to achieve their functions, such as accessing the network through network-related APIs,

obtaining user identifiers through telephony-related APIs, etc., which should also be recorded and reported by the sandbox.

Figure 5 illustrates the analyzability of the sample in different sandboxes and environments. It can be seen that the samples in **GP** have higher analyzability in virtual environments, and this is because Google Play has more consideration for compatibility with the apps, for example, they provide libraries for more CPU instruction set architectures (e.g. ARM, ARM64, and x86). On the contrary, malicious samples in **AZ** are generally not developed to meet specifications. To speed up development and reduce app size, they are usually only compatible with common CPUs and are not considered to run on virtual machines. Some sophisticated attackers may even block their apps from running in virtual machines to avoid dynamic detection. Besides, two sample datasets both have higher analyzability in real environments. This also highlights the importance of using real devices in dynamic analysis.

5.3.2 Content and format of reports

Before comparing the dynamic analysis capability and efficiency of DroidHook and DroidBox, it is necessary to introduce the content and format of their output reports. AndroPyTool's DroidBox module provides a JSON file for each normally running sample, and the objects contained in this file include:

- `apkName`: path and name of the sample file.
- `hashes`: hash values of the analyzed samples.
- `enfperm`: permissions requested by the sample (but not necessarily used).
- `opennet`, `closenet`, `sendnet` and `recvnet`: network operations.
- `cryptousage`: operations with the cryptography Android APIs.
- `sendsms`: operations for sending messages.
- `servicestart`: operations for starting services.
- `accessedFiles` and `fdaccess`: file read and write operations.
- `dataleaks`: leak of user's personal data.
- `dexclass`: loaded classes from DEX files.
- `recvactions`: intents that the sample responds.
- `phonecalls`: operations for making phone calls.

Although the classification of objects in the report is very detailed, we find DroidBox can not accurately detect every operation and has a lot of redundant content. On the other hand, DroidHook only outputs reports on operations of the Root APIs. This has many benefits: it reduces the pressure on subsequent analysis work and reduces storage space dependency. For example, in the report of a sample whose package name is "app.qrcode", there are 335 operations in `accessedFiles` of DroidBox: 206 for files in **/data** (and 204 for files in **/data/data/app.qrcode**), 31 for pipes, 94 for files in **/proc** and 4 for files in **/dev**. In `fdaccess`, the contents of file read and write operations are presented in hex format. Nevertheless, most of these are of little use. Files in **/data/data/app.qrcode** are usually used to store the sample's own data, and operations associated with them are quite normal. **/proc** is a virtual directory that provides an interface for access to the system kernel in the form of a file system.

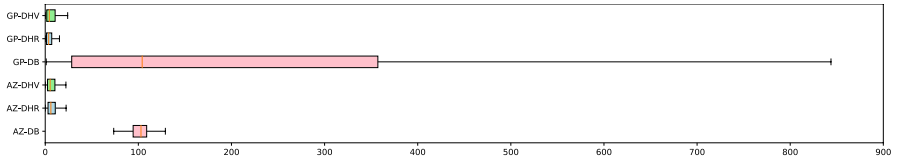


Fig. 6 Size of output reports for different sample datasets

Most of the operations in this category are to read the command line of a process. However, information about the process is unknown, which makes further analysis impossible. Another example is objects in `opennet`. DroidBox only provides the IP address and port of the destination, but these are often not enough. For example, it is not possible to know from this information whether the sample is trying to download an APK file for later installation.

DroidHook avoids these problems and makes every effort to ensure reports are accurate and detailed. Firstly, only Root API is monitored in DroidHook's reports. DroidHook does not provide information that is not related to Root APIs, such as permissions and hash values. There are better tools or methods to acquire them. Secondly, DroidHook can also extract the parameters of Root APIs. It is easy to know the URL accessed through network and the destination number and content of a short message. Finally, DroidHook is flexible and can conveniently add or remove APIs and parameters for analysis. Details and examples are provided in DroidHook's Github repository. All that needs to be known are the name of the package which the API belongs to and parameters to be monitored.

Figure 6 demonstrates the size of reports generated by **DB**, **DHV** and **DHR** for two sample datasets, and it can be seen that **DHV** and **DHR** require significantly less storage space. This is because, as mentioned above, DroidBox has a lot of redundant and irrelevant information in its reports. It is obvious that smaller report file size allows for more efficient follow-up analysis. Moreover, since the plots **DHV** and **DHR** correspond to generally do not differ much (although the maximum value of **DHV** is a little higher), we can conclude that, despite real devices have better compatibility than virtual environments, they have roughly the same detection capabilities for DroidHook.

Figures 7, 8 and 9 shows the amount of API call operations of two datasets detected by **DB**, **DHV**, and **DHR**, respectively. Through comparison, we can find the following three differences between DroidHook and DroidBox in terms of output reports. Firstly, DroidHook obviously detects more network-related operations. Secondly, DroidBox detects a large number of file-related operations. Thirdly, the difference between **DHR** and **DHV** is mainly in the detection of `TelephonyManager.getImei()` and `TelephonyManager.listen()`.

The three differences support our previous criticism of DroidBox. Firstly, from Fig. 7 we can find that network-related API operations are rarely detected by **DB**, while these samples do perform a large number of network operations. This is because that DroidBox is based on TaintDroid which only identifies tainted data transmitted through the network. The deficiency of network operation monitoring

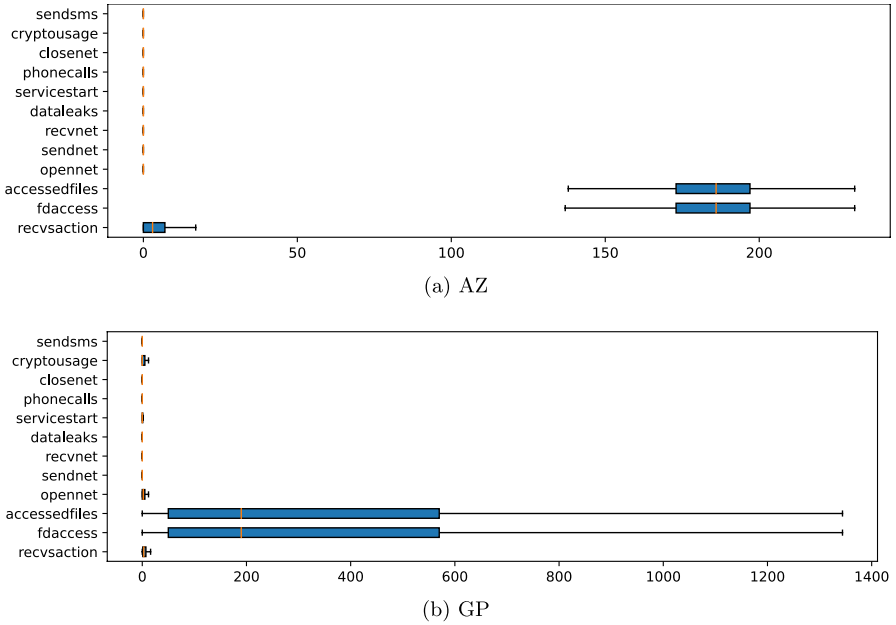


Fig. 7 Amount of operations detected by **DB**

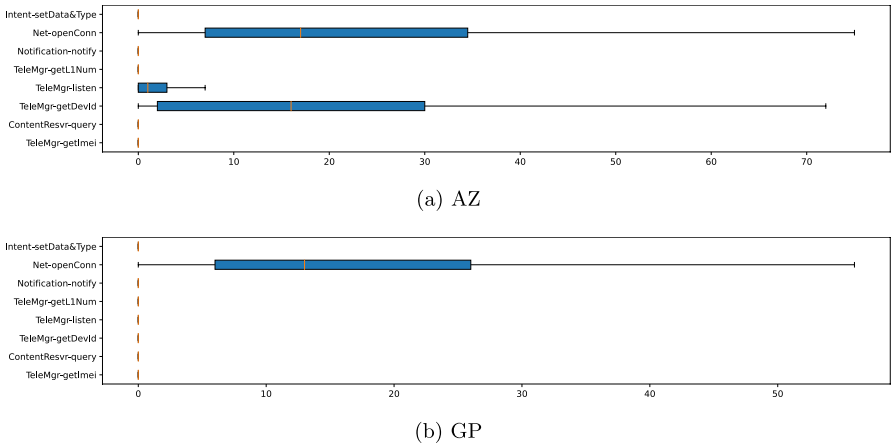
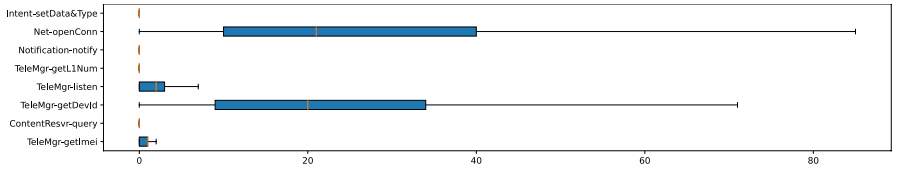
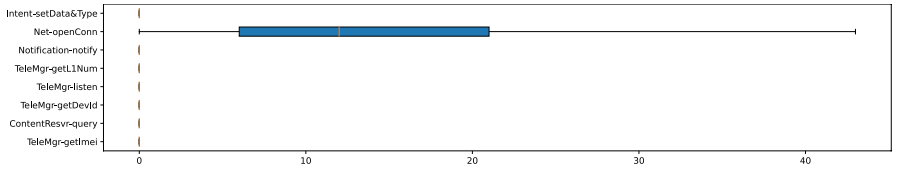


Fig. 8 Amount of operations detected by **DHV**

may cause the failure to detect malicious behaviors through network, such as privacy leaks and the download of ads, and a typical example is provided in Sect. 5.3.3. On the other hand, **DHV** and **DHR** detect more network operations, and the URI of each network operation is recorded in the log for further analysis. Secondly, the redundant report of DroidBox makes it difficult for subsequently analysis. It can be seen that **DB** outputs more than 600 monitoring logs about `accessedFiles` and



(a) AZ



(b) GP

Fig. 9 Amount of operations detected by **DHR**

`fdaccess` for nearly a quarter of samples in **GP** dataset, and these logs are mostly meaningless file descriptors with abstract data. For example, **DB**'s report has a large number of entries for file read of `/proc/[pid]/cmdline`. For Android, read this file to get the command line of the corresponding process, usually the package name of the application. The following analytical work in these reports is difficult. Thirdly, thanks to DroidHook's support for both virtual emulators and real devices, we can also observe the operation of the two sample sets in different environments. `TelephonyManager.getImei()` and `TelephonyManager.listen()` have a higher number of detections in real environments. This indicates that there may be behaviors that are hardly detected in the emulator, thus sandbox based on real devices is necessary. However, **DroidBox** does not support real devices and therefore might lead to evasion attacks.

From DroidHook's report, we can obtain the following observations. Firstly, comparing the observations of the two datasets, several APIs are barely detected. This might be because the Root API selected in our comparison experiment is not precise enough, and the Root API should be further determined after the application scenario of DroidHook is clear. Another possible reason is that the number of samples in the `AZ` dataset that perform the malicious operation in question is small. Secondly, for the malicious dataset **AZ**, **DHR** produces slightly more logs than **DHV**. Specifically, `TelephonyManager.getImei()` is almost undetectable in the virtual environment. This observation once again supports the idea that real devices may contribute to better detection.

5.3.3 A representative case

We discuss a representative case to show how DroidHook compares to DroidBox. The sample comes from **AZ**, and is flagged by five security vendors as riskware or potentially unwanted app, according to VirusTotal's report. The SHA-1 value of this sample is 2CECAFEC9B454DFF39991EFC036CACA521B9754B. Both

DroidHook and DroidBox generate longer reports (6.1K and 128K, both above the median), indicating that this sample exhibits more sensitive behavior. In addition, during the analysis, we find that this sample has the typical behavior of collecting device information and sending it to a remote server. For these reasons, we believe this sample can be used as a representative sample for further discussion. In the following, we will discuss the reports of DroidHook and DroidBox for this sample separately and compare their differences.

DroidHook For the specified API to be monitored, DroidHook outputs a total of 54 log entries, including 29 entries about `getDeviceId()`, 24 entries about `openConnection()` and 1 entry about `getImei()`. Based on the timestamp, we can clearly find the order in which this sample performs sensitive behaviors. In addition, since DroidHook can also monitor the API call parameters, we can also observe the specific data of the sample leak, get the sample destination URL and further analysis. Specifically, DroidHook reports that this sample sent data such as OS type (Android), device model (ONEPLUS A3000), network type (Wi-Fi), MAC address, IMEI, etc. to a URL after calling a series of sensitive APIs to obtain device information. Afterwards, it visits a URL in large numbers, which is subsequently considered to belong to a data analytics service provider. Based on the information provided by DroidHook, we can accurately determine that the sample has sensitive behavior and clearly see the procedure of data leakage.

DroidBox DroidBox outputs a total of 512 log entries, including 232 entries about `fdaccess`, 231 entries about `accessedFiles`, 25 entries about `dexclass`, 22 entries about `servicestart` and 2 entries about `dataleaks`. Since DroidBox is based on TaintDroid, it can detect IMEI leaks through taint tracking. However, DroidBox does not provide further information, such as where the IMEI was sent to. Moreover, DroidBox does not detect other information leaks. In addition, DroidBox records a large number of pipeline-related operations and data operations in the `/proc/[pid]/cmdline`, and such a log is not clear enough to see the logical sequence of sensitive behavior nor to rely on it for further analysis.

6 Conclusion

In this paper, we illuminate the risk of Android malicious apps and propose a novel sandbox for Android malware dynamic analysis named DroidHook. Our goal is to provide a lightweight, compatible, easily expandable and automated sandbox for Android security researchers and practitioners to monitor Root API calls. Therefore, DroidHook supports both emulators and real devices, strives for ease of expansion, and can work with many versions of Android. In future work, we aim to test and optimize DroidHook's performance in more realistic detection environments and enhance the granularity of reports for a wider range of application scenarios.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments and suggestions.

Author Contributions YC wrote the original manuscript text. YS and ZL reviewed and edited the final version of the manuscript. All authors prepared and conducted experiments.

Declarations

Conflict of interest The authors declare no competing interests.

References

Aafer, Y., Du, W., Yin, H.: Droidapiminer: mining API-level features for robust malware detection in android. In: International Conference on Security and Privacy in Communication Systems. pp. 86–103. Springer, Cham (2013)

Alzaylaee, M.K., Yerima, S.Y., Sezer, S.: Dynalog: an automated dynamic analysis framework for characterizing android applications. In: 2016 International Conference on Cyber Security and Protection Of Digital Services (Cyber Security), IEEE, pp. 1–8 (2016)

Alzaylaee, M.K., Yerima, S.Y., Sezer, S.: Emulator vs real phone: android malware detection using machine learning. In: Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics, pp. 65–72 (2017)

Arp, D., Spreitzenbarth, M., Hubner, M., et al.: Drebin: effective and explainable detection of android malware in your pocket. In: NDSS, pp. 23–26 (2014)

Arshad, S., Shah, M.A., Wahid, A., et al.: Samadroid: a novel 3-level hybrid malware detection model for android operating system. IEEE Access **6**, 4321–4339 (2018)

AV-TEST: Malware statistics and trends report. https://www.av-test.org/en/statistics/malware/ (2020). Accessed 06 Oct 2020

Cai, H., Meng, N., Ryder, B., et al.: Droidcat: effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. **14**(6), 1455–1470 (2018)

Cai, H., Meng, N., Ryder, B., et al.: Droidcat: effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. **14**(6), 1455–1470 (2019)

Chang, W.L., Sun, H.M., Wu, W.: An android behavior-based malware detection method using machine learning. In: 2016 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC), IEEE, pp. 1–4 (2016)

Chen, X., Li, C., Wang, D., et al.: Android HIV: a study of repackaging malware for evading machine-learning detection. IEEE Trans. Inf. Forensics Secur. **15**, 987–1001 (2019)

Cho, H., Yi, J.H., Ahn, G.J.: Dexmonitor: dynamically analyzing and monitoring obfuscated android applications. IEEE Access **6**, 71229–71240 (2018)

DroidBox. Droidbox: Dynamic analysis of android apps. https://github.com/pjlantz/droidbox (2020). Accessed 07 Oct 2020

Enck, W., Gilbert, P., Han, S., et al.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 1–29 (2014)

Fan, M., Liu, J., Wang, W., et al.: Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. IEEE Trans. Inf. Forensics Secur. **12**(8), 1772–1785 (2017)

Fan, M., Liu, J., Luo, X., et al.: Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Trans. Inf. Forensics Secur. **13**(8), 1890–1905 (2018)

Feng, P., Ma, J., Sun, C., et al.: A novel dynamic android malware detection system with ensemble learning. IEEE Access **6**, 30996–31011 (2018)

Gajrani, J., Agarwal, U., Laxmi, V., et al.: Espydroid+: precise reflection analysis of android apps. Comput. Secur. **90**(101), 688 (2020)

Gao, H., Cheng, S., Zhang, W.: Gdroid: android malware detection and classification with graph convolutional network. Comput. Secur. **106**(102), 264 (2021)

Kelly, G.: Report: 97% of mobile malware is on android. this is the easy way you stay safe. https://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/ (2014). Accessed 06 Oct 2020

Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., et al.: Andrubis—1,000,000 apps later: a view on current android malware behaviors. In: 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), pp. 3–17 (2014)

Martín, A., Lara-Cabrera, R., Camacho, D.: Android malware detection through hybrid features fusion and ensemble classifiers: the andropytool framework and the omnidroid dataset. Inform. Fusion **52**, 128–142 (2019)

Nicheporuk, A., Savenko, O., Nicheporuk, A., et al.: An android malware detection method based on CNN mixed-data model. In: ICTERI Workshops, pp. 198–213 (2020)

Onwuzurike, L., Mariconti, E., Andriotis, P., et al.: Mamadroid: fetecting android malware by building Markov chains of behavioral models (extended version). ACM Trans. Privacy Secur. (TOPS) **22**(2), 1–34 (2019)

Peiravian, N., Zhu, X.: Machine learning for android malware detection using permission and API calls. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, IEEE, pp. 300–305 (2013)

Samhi, J., Gao, J., Daoudi, N, et al.: Jucify: a step towards android code unification for enhanced static analysis. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1232–1244 (2022)

Sihag, V., Vardhan, M., Singh, P., et al.: De-lady: deep learning based android malware detection using dynamic features. J. Internet Serv. Inf. Secur. **11**(2), 34–45 (2021)

Statista.: Global mobile OS market share 2012-2022. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (2022). Accessed 06 Oct 2022

Sugunan, K., Kumar, T.G., Dhanya, K.: Static and dynamic analysis for android malware detection. In: Advances in Big Data and Cloud Computing, pp. 147–155. Springer, Berlin (2018)

Sun, M., Wei, T., Lui, J.C.: Taintart: a practical multi-level information-flow tracking system for android runtime. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 331–342 (2016)

Sun, X., Li, L., Bissyandé, T.F., et al.: Taming reflection: an essential step toward whole-program analysis of android apps. ACM Trans. Softw. Eng. Methodol. (TOSEM) **30**(3), 1–36 (2021)

Tam, K., Fattori, A., Khan, S., et al.: Copperdroid: automatic reconstruction of android malware behaviors. In: NDSS Symposium 2015, pp. 1–15 (2015)

Tian, K., Yao, D., Ryder, B.G., et al.: Detection of repackaged android malware with code-heterogeneity features. IEEE Trans. Dependable Secure Comput. **17**(1), 64–77 (2017)

Wang, W., Zhao, M., Wang, J.: Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J. Ambient. Intell. Humaniz. Comput. **10**(8), 3035–3043 (2019)

Xiao, X., Zhang, S., Mercaldo, F., et al.: Android malware detection based on system call sequences and LSTM. Multimedia Tools Appl. **78**(4), 3979–3999 (2019)

Xu, L., Zhang, D., Alvarez, M.A., et al.: Dynamic android malware classification using graph-based representations. In: 2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud), IEEE, pp. 220–231 (2016)

Xue, L., Qian, C., Zhou, H., et al.: Ndroid: toward tracking information flows across multiple android contexts. IEEE Trans. Inf. Forensics Secur. **14**(3), 814–828 (2018)

Zheng, M., Sun, M., Lui, J.C.: Droidtrace: a ptrace based android dynamic analysis system with forward execution capability. In: 2014 International Wireless Communications and Mobile Computing Conference (IWCMC), IEEE, pp. 128–133 (2014)

Zungur, O., Stringhini, G., Egele, M.: Libspector: Context-aware large-scale network traffic analysis of android applications. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, pp. 318–330 (2020)