# Towards mitigating blindspots in Android sandbox approaches for detecting malware

Anonymous Author(s)

## ABSTRACT

Android is by far the most popular operating system for the mobile platform and the ubiquitous nature of smartphones in our daily lives has only made its security a significant topic for researchers and practitioners alike. Previous research has shown that security experts can benefit from the mining sandbox approach to classify malware. The Android OS exposes several sensitive APIs that allow apps to gain access to user's sensitive resources like contacts, locations and call logs. Nonetheless, in the literature, we did not find a quantitative assessment that characterizes malicious behavior, in terms of calls to sensitive APIs. Moreover, previous research reports that the mining sandbox approach has some limitations (a false negative rate around 35%).

To complement prior studies and mitigate the limitations of mining sandbox approaches, we address both issues in our paper. First, we carefully investigate the calls to sensitive APIs the malicious apps introduce. From our investigation of 800 repackaged malicious apps, we found that just 14 sensitive APIs from a list of 162 were injected into most repackaged malicious apps. This result might help the research community focus their attention on frequently abused sensitive privileges. Second, we explore two techniques that complement the mining sandbox approach and improve the false negative rates: a dynamic call trace assessment that helps identify suspicious paths from app entry points to sensitive API calls, and a simple, yet effective, analysis of Manifest files. Using these new techniques reduces the number of false negatives by a factor of more than 20%.

## KEYWORDS

Malware Detection, Mining Sandboxes, Software Security, Android Platform, Empirical Studies

**ACM Reference Format:**

Anonymous Author(s). 2022. Towards mitigating blindspots in Android sandbox approaches for detecting malware. In *Proceedings of The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn



*ESEC/FSE 2022, 14 - 18 November, 2022, Singapore*



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Almost two-thirds of the world use mobile technologies, such as smartphones and tablets, acquired a central role in everyday life in the last decade [1][2]. In this context, Android Operation System has dominated this market, with around 150 billions of Android application [1] (apps) downloaded by October 2018 KN: Do we have something more new from its marketplaces (Google Play) [3]. Due to this growing popularity, we witnessed an unprecedented growth of Android apps that perform malicious activities (malware). This inturn has made security issues in Android apps a relevant research topic. Several techniques have emerged to identify malicious behavior and vulnerabilities in Android apps, such as static analysis algorithms to expose private information leaks [4], or reveal misuse of cryptographic primitives. [5]

Another alternative for protection from Android malicious behavior consists in the use of dynamic analysis to mine Android sandboxes [6]. Such approaches use automated testing tools (i.e., dynamic analysis) to explore apps behavior in terms of access to sensitive resources. A particularly popular way of creating malware is to injecstart with a benign version of an app from an official app store such as Google Play and inject it with code performing malicious activities such as broadcasting credit card information to a private server (KN: Here add a citation that shows prevalence of repackaged malware). The state of the art in terms of accuracy in detecting such malware is DroidBot [7] which has an accuracy rate of X%. But their evaluation was performed on a set of 102 app pairs [2]. Our internal analysis also revealed that the similarity index between the app pairs used in DroidBot's evaluation was 77%. Previous studies from David Lo et al. [8] has revealed that malwares tend to be harder to detect if the similarity index is lower. Both the above mentioned points indicate that the relatively high accuracy reported in earlier evaluations of state of the art mining sandbox approaches may not scale to larger datasets with a more diverse and potentially higher similarity index.

In this paper, we address two open questions that arise. First, we seek to understand what the accuracy of the state of the art in mining sandbox approaches, DroidBot is when presented with a larger dataset (800 as opposed to the 102). This new dataset also has a combined average similarity index of $Y\%$ which is lower than the original $77\%$. Secondly, if the accurary rate has indeed significantly dropped during the reproduction of DroidBot with a more real-world representative dataset what approaches can improve upon the accuracy.

[1] In this paper, we will use the terms Android Applications, Android Apps and Apps interchangeably, to refer to Android software applications

[2] App pairs here refers to pairs of the same app containing a benign and a malicious repackaged version

To answer the first question, we present the results of a study, where we observed the reults of DroidBot on a set from 800 real apps pairs (benign/malicious). For this reproduction study, we take advantage of DroidXP [9], a tool suite that supported us not only to integrate DroidBot but also in our study setup and data collection.

To answer the second question, we hypothesize that mining sandbox approaches such as DroidBot perform a superficial analysis of differences between app pairs, i.e., only the difference in the set of sensitive APIs called. We assert that there are two major blindspots in the state of the art:

(1) The differences between two apps that call the same set of sensitive APIs, but differ in their dynamic call traces between the app's entry point and the sensitive APIs.
(2) The differences in the requested permissions between the two variant of apps.

To this end, we used an auxiliary tool from DroidXP project called DroidXPTrace to compare all traces from an entry point to sensitive APIs call and checking if they are divergences. To observe the differences in the manifest files we harness the standard apk analysis tool from Android [10].

In particular, we investigated the following specific questions in our study: (answered in Section 4):

(RQ1) How well does the state of the art in mining sandbox approaches, DroidBot perform with a large dataet of app pairs with a diverse similarity index?
(RQ2) Numerically, how relevant is a dynamic call trace analysis to improving malware detection in support of the mine sandbox approach?
(RQ3) Numerically, how relevant is a manifest files analysis to improving malware detection in support of the mine sandbox approach?

Our findings indicate that the accuracy of DroidBot significantly drops (to $Z\%$) when reproduced on a larger dataset of 800 app pairs with a similarity index of $53.3\%$. Our experiments also reveal that the accuracy of sandbox approaches improve when they are made aware of the differences in the dynamic call trace and difference in permissions requested using the manifest file. Specifically, ... KN: Please add here the results of path analysis and manifest files individually and their intersection During our analysis we also observed another interesting insight that out of the 162 sensitive APIs explored, just xx (xx%) was injected at most repackage apps. It indicates that we have a small set of sensitive APIs used generally by malicious developers, pointing out that researchers can concentrate their effort on this specific set of sensitive APIs to improve the security of Android apps. In summary, this paper makes the following contributions:

1. A reproduction of the state of the art in mining sandbox approaches, Droidbot scaled in terms of number of app pairs and similarity index.
2. A broad comprehension about the role of trace analysis and static analysis on Android manifest file in improving the accuracy Android sandbox approach.
3. A in depth look into the kind of sensitive APIs that plague most repackaged apps.
4. A reproduction package of the studies online. Scripts for statistic analysis are also available.[3]

The rest of the paper is organized as follows.....KN: Here, please describe each section in one sentence

## 2 BACKGROUND AND RELATED WORK

The Android bytecode language [11] favors reverse engineer tasks. That is, software developers can easily reverse-engineer real apps (benign), modify their contents by inserting malicious code (malware), repackage them with the malicious payloads, and re-advertise them in the official market, Google Play Store, or other markets. Repackage Android apps can leverage the popularity of real apps, to increase its propagation and spread malware. Repackaging has been raised as a great security problem in Android ecosystem by stakeholders in the app development industry and researchers. There are works [12] claiming that about 25% of Google Play Store app content is repackaging apps. Nevertheless, all the workload to detect and remove malware from markets by the stores (official and no official), have not been accurate enough to address the problem as fast as possible. As a result, repackaged Android apps follow threatening security and privacy of unsuspicious Android app users, beyond compromising the copyright of the original developer [13]. Aiming at mitigating this threat, static and dynamic analysis approaches have been proposed.

### 2.1 Dynamic and Static Analysis on Android apps

There is a large body of work that explores the use of program analysis techniques to detect malware.

Several works have been proposed to detect malware based on sensitive method calls and permission control [14–16]. Cai et al. [17] presented a longitudinal study on Android apps focusing on run-time behaviors. However, this work does not focus specifically on malware detection but on general security gaps in apps by considering only benign apps. Fangfang et al. [18] proposed ViewDroid, which models the UIs of Android apps as a directed graph. Although ViewDroid also works by comparing app pairs to identify repackaged apps, their focus is UI centric.

On static analysis approaches exploring Android Manifest files, Kim et al. proposed RomaDroid [13]. Their approach does not consider the structural context in Manifest files, but rather treat the files as sequence of strings and perform a lowest common subsequence (LCS) based approach to detect repackaged apps. Au el al. [19] also apply static analysis on Android Manifest files to detect vulnerabilities in Android apps. They do this by mapping requested permissions to sensitive API calls in the code.

Li et al. [20] provided a systematic knowledge on Android malware by conducting an empirical study comparing malicious repackage app with their benign counterparts (1,497 app pairs). They found that the majority of Android malware are repackaged versions of benign apps that do not do anything complex modifications, many times simply reusing library code.

[3] https://github.com/droidxp/paper-droidxp-replication.git

In the domain of detecting repackaged apps by comparing app pairs, Crussell et al. [21] proposed DNADroid, which compares program dependence graphs, and Zhou et al. [22] DroidMoss which detects and analyzes repackaged apps using a fuzzy hashing technique.

## 2.2 Mining Android Sandboxes

A *sandbox* is a well-known mechanism to secure a system and forbid a software component to access resources that it is not allowed to. Sandboxes have also been used to build an isolated environment on an electronic device within which applications cannot affect other programs, the network, or other device data [23]. In this scenario, the idea of using sandboxes emerged from the need to testing unsafe software, possible malware, without worrying about the integrity of the device under test [24], shielding the operation system from any security issue. To ensure safety, a sandbox environment should have the minimum requirements to run the program (make sure the program will not escape the sandbox), and make sure it will never assign the program greater privileges than it should have, working with the principle of *least privilege*. This principle ensures unauthorized access to resources, improving the system's overall health. Regarding Android ecosystem, the principle of the *least privilege* is ensured by sandboxing too, where apps never access direct resources or data of other apps. Access to sensitives resources like contacts list is granted through specific APIs (Application Programming Interface), which are managed by permissions [25].

The Mining Android Sandbox approach [6] aims at automatically building a sandbox from dynamic analysis (i.e., using automatic test generation tools). The main idea is to explore apps based on their calls to sensitive APIs. Thus, sandboxes build upon these calls to create safety rules and then block future calls to other sensitive resources, which diverge from those found in the first exploratory phase. Using a test generation tool named Droidmate [26], Jamrozik et al. [6] proposed the first mainstream implementation of the Mining Android Sandbox approach, called Boxmate. Boxmate records the occurrences of calls to sensitive APIs and the event that triggers these calls, like a button click. It is possible to configure Boxmate to record events associated with each sensitive call as tuples (event, API). Jamrozik et al. argue that, in this way, Boxmate generates finer granularity results which might reduce false alarms, even with the presence of reflection which is quite commonly used in malicious apps [27]. Figure 1 summarises the process of mining sandbox.

The Mining Android Sandbox approach accuracy suggests a close relationship with the efficiency of the exploratory phase. The more efficient the test generator tools (for instance, in terms of code coverage), the more accurate would be the resulting sandbox. Besides being used to generate Android sandboxes, the Mining Android Sandbox approach is also effective to classify malwares [8]. In this scenario, the *effectiveness* of the approach is estimated in terms of repackage/malware identification. Previous studies [8, 9] that explored the effectiveness of mine sandboxes, investigating and comparing the performance of different test generator tools, with diverse exploratory strategies.

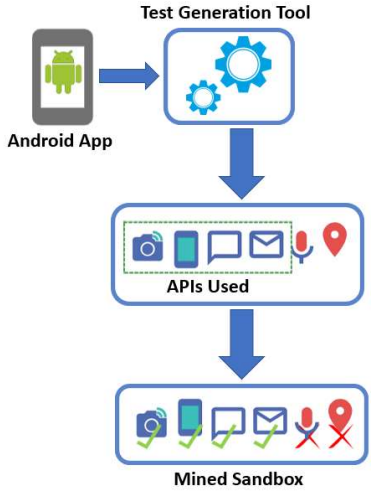


**Figure 1: Mine Sandbox.**

## 2.3 The Mining Android Sandbox Approach for Malware Identification

The focus of our paper is in approaches that mine android sandboxes to classify Android Malware. There is a vast body of research in this direction. Bao et al. [8] conducted an empirical study to investigate the effectiveness of mining android sandbox approaches for detecting malware, exploring test generation tools, including Droidmate. The authors found that, in general, the sandboxes constructed by test generator tools can detect more than 70% malicious apps in a dataset comprising 102 pairs (benign/malicious). The study also presented that among 5 test generation tools used, DroidBot [7] leads to the most effective sandbox. Le et al. [28] extend the work of Bao et al. by combining more categories of sensitive APIs, and also considering the impact of actual arguments on the sandboxes definition. Costa et al.[29] investigated the impact of static analysis to complement the accuracy of dynamic analysis tools for mining sandboxes, in terms of malware detection. The study found that the static analysis alone could detect almost half of repackage apps in a dataset of 96 pairs.

Our work, although closely related to previous studies, differs from them in several aspects. We present a broader assessment of the effectiveness of the mining sandbox approach for malware identification, resulting in the first in depth characterization of the calls to sensitive APIs that frequently appear in the repackage version of the apps. Our assessment is also more comprehensive: instead of considering 102 pairs of benign/malign apps, we execute our study consider 800 pairs of apps. We also explore two possible strategies to complement the mining sandbox approach for malware identification. The first explores the impact of dynamic call graphs through the comparisons of traces from entry point to the calls to sensitive

APIs that result from the executions of the benign and malign versions of an app. The second explores suspicious change patterns in the Android Manifest file of the apps, using a simple, yet effective, static analysis approach.

# 3 EXPERIMENTAL SETUP

In this section, we describe the setup of our study. First we describe how we mined the data-set of Android app pairs that will serve as a benchmark for our study (Section 3.1). Then, we describe the setup of our infrastructure used to perform the study(Section 3.2) followed by a discussion on the hardware setup and configurations used (Section 3.3). We finally describe the adaptations we performed on the base setup in order to facilitate the individual analyses (Section 3.4, 3.5, 3.6)

## 3.1 Constructing our dataset

Most malware are simply repackaged versions of benign apps from the official store that injects code performing malicious tasks. Zhou et al. [30] curated a wide-ranging malware collection, where 80% of samples are know to be built by repackaging benign apps. Because of this our dataset is composed by app pairs (benign-malicious). KN: I am not sure how the Zhou's work is the reason we should have app pairs. Did they mention in their study that app pairs are needed. Or is this the basis of our app pairs. Can we simply not justify the need for pairs by saying something in the lines of "We need both versions of a repackaged app to test the accuracy of our tool?"

To curate our app pairs, we used Androzoo [31], a collection of Android apps mined from multiple markets, including the official Google Play [4]. Androzoo makes an ideal source of apps as it was designed with a primary goal of aiding research studies such as ours.

Due to space and time constraints, we capped our download of apps to 20 hours and 42 GB of apps. This resulted in 7268 app pairs. Androzoo sometimes contains multiple malicious versions for the same benign app. In such cases, we picked the first malicious version ensuring that the resulting pairs were unique in terms of their original apps. This step resulted in our dataset containing 1831 app pairs. Among 1831 initial pairs, only 1395 apps could be successfully instrumented by DroidFax. Among the 1395 instrumented pairs, we had installation errors on our emulator (Pixel 4 running Android API 28), many times due to compatibility issues.

In the end, we obtained 824 valid app pairs that could be used for our study. For time issues, we decided to use just 95 KN: This number needs to be changed to our final set pairs at our research. Figure 2 presents the markets where the most of malicious version apps were collected according to Androzoo including several from the Google play store.

KN: I have removed the following as it is inconsistent with our current story. We no longer performn just a study, we offer an approach. And we no longer study all four, only the top one Droidbot and built on top of it. In order to understand the limitations in accuracy of mining sandbox approaches, we perform our study and build upon DroidBot [7] which is considered as the state of the art in mining sandbox approaches in terms of accuracy.

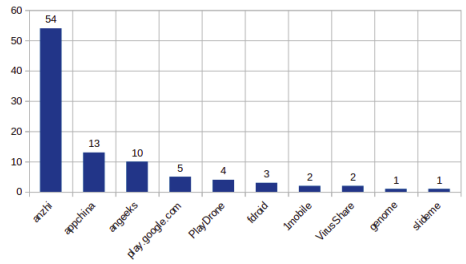


**Figure 2: Markets where malware was downloaded.**

## 3.2 Infrastructure

KN: We currently only use one test generation tool. The following paragraph needs to be adjusted to reflect that For our study, we require an extensive infrastructure that will allow us to integrate different test case generation tools. To this end, we used and extended DroidXP[9], a benchmark tool that allows researchers and developers, to integrate and compare test case generation tools for mining sandboxes.

Using DroidXP, test case generation tools can be integrated and deployed easily [5]. Another reason to use DroidXP is that it relies on DROIDFAX [32], which allowed us to instruments Android apps and collect important information about their execution. For example, this can be used to query for the calls to the sensitive APIs from an app during executions which is important to monitor malicious behavior.

Benchmarking using DroidXP typically happens in three phases:

**Phase 1: Instrumentation:** In the first phase, a set of APK files is provided as input to DroidXP. This set is composed of all the app pairs (benign/malicious) used in our study. Each APK file is then instrumented to be able to collect data about the execution. This instrumentation is performed using DroidFax. To improve the performance across multiple executions, the set of pairs already instrumented are made available. In this phase, each app is statically analyzed to collect the number of methods and classes inside the app, which is necessary to measure coverage during execution.

**Phase 2: Execution:** In this phase, the execution is done by deploying an instrumented APK file in an Android emulator and executing the instrumented app using one or more test generation tools for a chosen period of time. To ensure that each execution gets the benefit of running on a fresh android instance without biases that could stem out of history, all data

[4] https://play.google.com/store

[5] The DroidXP benchmark is available at https://github.com/droidxp/benchmark

stored on the emulator from previous executions are wiped out from the emulator.

**Phase 3: Result Analysis:** During the execution, all the data that is required to compute the results are persisted by DroidXP, using Logcat [33], one of the Android SDK's native logging tools. This can be used to perform analysis on the results of executing test generators on apps and observe malicious behavior.

**DroidXPTrace**. To leverage our study, we also used an auxiliary tool from DroidXP project called DroidXPTrace [6]. DroidXPTrace allowed us to monitor aspects like dynamic call graphs which was not possible using vanilla version of DroidXP. DroidXPTrace, using the result from Phase 3 of DroidXP, creates a call graph of app pair, exploring traces from its entry point to sensitive API access. In the end, it compares both call graphs (benign and malicious app version), and generated a JSON file recording the following information:

- `benign`: Name of log file from the benign app version
- `malign`: Name of log file from the malicious app version
- `benignGraphs`: call graphs that contain traces with access to sensitive methods in the benign version of the app.
- `malignGraphs`: call graphs that contain traces with access to sensitive methods in the malicious version of the app.
- `methodsAccessedOnlyByMalign`: The sensitive methods that are accessed only by the malicious version of the app. This information is important to identify if a particular sensitive call that is undetected by a sandbox occurs only in a malicious version.
- `benignGraphContainsMalignGraph`: Comparison between benign and malicious sub-callgraphs
- `hasDifferentTraces`: whether the benign and malicious app version have different traces to sensitive resources.

This information helped us to explore traces between an app's entry point and its access to sensitive methods during run time. KN: What I have commented below is talking about the study itself and not the setup. Please ensure other such instances are removed if I have missed them.

## 3.3 Hardware and procedure setup

We deployed our experiment on a 32-Core, AMD EPYC 7542 CPU, 512 GB RAM, storage Samsung SSD 970 EVO 1TB machine running a 64-bit Debian GNU/Linux 11. We also configured our emulator to run all selected apps on Google Android version 9.0, API 28, 512M SD Card, 7GB internal storage, with X86 ABI image.

For our study, we configured DroidXP to run each of the 95 KN: This number needs to be changed here too. I suggest introducing a macro for this and using it everywhere to make sure everytime our experiment evolves, we dont need to change this everywhere app pairs using Droidbot testing tool for three minutes. To mitigate noise, we repeated the full process three times which took in total ($95 \times 2$ apps (benign/malicious) $\times 3$ runs $\times 3$ min) + ($95 \times 2$ apps $\times 1.5$ min for emulator reboot) $\approx$ 34 machine hours.

Although it was possible to run more than 10 emulators in parallel on one physical machine, to avoid any interference resulting from context switching within the operating system, we chose to run one emulator at a time. Hence, all evaluation processes took around 3 days and additional 5 days for environment deployment.

In the following subsections, we describe the setup required to perform custom analysis that were central to our study.

## 3.4 Sensitive API setup

FH: Here the text talk about false positive. We have to change to talk about how we extract the most sensitive APIs used by malicous apps KN: I am ignoring this subsection for review as it is no longer relevant. Please add some details about the sensitive APIs collection so it can be extended upon. To perform the false positive analysis we used the infrastructure of DroidXP, described in section 3.2. After finishing **Phase 2 (execution)**, we have all explorer information about the apps under analysis by Droidbot test generation tools.

Our analysis concert just at the benign version of apps, since false positive alarm could occur when some expected behavior from the benign app version is not seen during mining step. Hence, as a first step we collect all sensitive methods (SM) by the union of three execution of mine sandbox at all benign apps version. With this step, we could have a base set of sensitive methods called by each benign version from all 95 apps explored in our study. After, we executed mine sandbox again at the same set of benign apps version seven times. At each execution we observed the set of sensitive methods called, and compare them with the initial set of sensitive methods composed by the union of methods explored by the first three executions. In the end, we executed 7 tests, checking if the difference between all the these sets are empty sets, as below:

$$\text{(Test 1)}\ \{\{SM01\} \cup \{SM02\} \cup \{SM03\}\}\ \mathit{diff}\ \{SM04\} = \emptyset$$
$$\text{(Test 2)}\ \{\{SM01\} \cup \{SM02\} \cup \{SM03\}\}\ \mathit{diff}\ \{SM05\} = \emptyset$$
$$\text{(Test 3)}\ \{\{SM01\} \cup \{SM02\} \cup \{SM03\}\}\ \mathit{diff}\ \{SM06\} = \emptyset$$
$$\text{(Test 4)}\ \{\{SM01\} \cup \{SM02\} \cup \{SM03\}\}\ \mathit{diff}\ \{SM07\} = \emptyset$$
$$\text{(Test 5)}\ \{\{SM01\} \cup \{SM02\} \cup \{SM03\}\}\ \mathit{diff}\ \{SM08\} = \emptyset$$
$$\text{(Test 6)}\ \{\{SM01\} \cup \{SM02\} \cup \{SM03\}\}\ \mathit{diff}\ \{SM09\} = \emptyset$$
$$\text{(Test 7)}\ \{\{SM01\} \cup \{SM02\} \cup \{SM03\}\}\ \mathit{diff}\ \{SM10\} = \emptyset$$

We observed if all execution explored the same sensitive methods at each benign app in our dataset (95 app pairs). We consider that occur a false positive alert when at least one of seven tests fail, i.e. if one of the 7 tests returns a non-empty set.

As a second step, we also choose a random execution, and check if the set of sensitive methods collected by this execution, is the same of other 9 sensitive methods set, collected by others 9 executions. At this second step, the fifth execution was the choose one as a base execution for tests, as described below:

$$\text{(Test 8)}\ \{SM05\}\ \mathit{diff}\ \{SM01\} = \emptyset$$
$$\text{(Test 9)}\ \{SM05\}\ \mathit{diff}\ \{SM02\} = \emptyset$$

[6] The DroidXPTrace is also available at https://github.com/droidxp/droidxptrace

(Test 10) $\{SM05\}\ diff\{SM03\} = \emptyset$
(Test 11) $\{SM05\}\ diff\{SM04\} = \emptyset$
(Test 12) $\{SM05\}\ diff\{SM06\} = \emptyset$
(Test 13) $\{SM05\}\ diff\{SM07\} = \emptyset$
(Test 14) $\{SM05\}\ diff\{SM08\} = \emptyset$
(Test 15) $\{SM05\}\ diff\{SM09\} = \emptyset$
(Test 16) $\{SM05\}\ diff\{SM09\} = \emptyset$

As the first analysis, we also consider a false negative occurrence, if one of the 9 remaining tests returns a non-empty set.

## 3.5 Trace analysis

As described at section 3.2, we take advantage of DroidXP's auxiliary tool, DroidXPTrace to create dynamic call graphs of each app pair (benign/malicious), as executed by Droidbot test generation tool. With these call graphs, we investigated specifically those app pairs that were not described as a malware during the exploratory step, i.e, the test generation tool DroidBot collected the same set of sensitive APIs for both version. If a dynamic call graph of these app pairs presented different traces from entry point to sensitive APIs call at both versions, we suspect this to indicate presence of malware.

Figure 3 present a example of benign and malicious call graph. Although both app version access the same set of sensitive resources, malicious version follows a different execution trace.

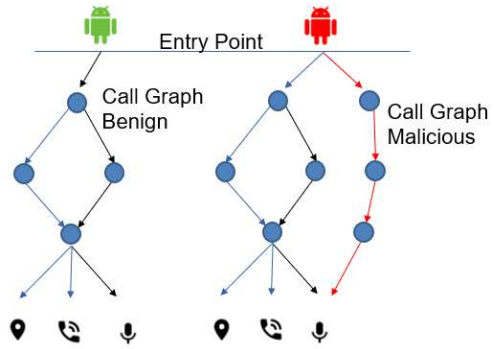


**Figure 3: Benign and Malicious call graph.**

## 3.6 Manifest file analysis

Our work also observed the Manifest file of all app pairs. These files present essential security information which the Android system must make available before executing an app. Among other things, a list of requested permissions and a list of important components are persisted in the manifest file. Unfortunately, Manifest files are generally not considered by sandbox approaches when detecting malware inspite of the fact that they can are easily modified by malicious developers [34]. For instance, by inserting new permission requests or component capabilities (actions). Such injections can happen either manually or through automated scripts.

Automated process sometimes generate Manifest file with duplicate permission requests, as the original app may already contain the permission request. Such duplication also happen when repackaged apps add a component with a capability which was requested by another component. On top of this, our manifest analysis also allowed us to monitor suspicious new permissions or excessive amounts of permissions.

To extract and analyze the Manifest file, we used an in-house Android SDK analyzer called *apkanalyzer*. We implemented a python script that used *apkanalyzer* and computed which malicious apps had duplicated request permission or duplicated actions. The script also extracted how many permissions were requested by all apps, to help us monitor potential suspicious features. Listing 1 and Listing 2 present an example of duplicated permission extracted from malicious version of the app **[com.ifeel.frogjump]**, and an example of duplicated component capabilities from malicious version of the app **[com.koushikdutta.superuser]** respectively:

**Listing 1: Example of duplicated permission from malicious version of app (com.ifeel.frogjump)**

```
13:M >  <uses-permission
        android:name="android.permission.READ_PHONE_STATE" />

16:M >  <uses-permission
        android:name="android.permission.ACCESS_COARSE_LOCATION" />

19:M >  <uses-permission
        android:name="android.permission.INTERNET" />

22:M >  <uses-permission
        android:name="android.permission.ACCESS_NETWORK_STATE" />
    .
    .
    .
134:M > <uses-permission
        android:name="android.permission.INTERNET" />

137:M > <uses-permission
        android:name="android.permission.WAKE_LOCK" />

140:M > <uses-permission
        android:name="android.permission.READ_PHONE_STATE" />

143:M > <uses-permission
        android:name="android.permission.ACCESS_NETWORK_STATE" />

146:M > <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

149:M > <uses-permission
        android:name="android.permission.ACCESS_WIFI_STATE" />
```

**Listing 2: An example of duplicated component capability from malicious version of app (com.koushikdutta.superuser)**

```
101:M > <receiver
102:M >  android:name=".SuCheckerReceiver">
103:M >  <intent-filter>
104:M >    <action
105:M >      android:name="android.intent.action.BOOT_COMPLETED" />
106:M >  </intent-filter>
107:M > </receiver>
108:M > <receiver
109:M >  android:name=".PackageChangeReceiver">
110:M >  <intent-filter>
111:M >    <action
112:M >      android:name="android.intent.action.BOOT_COMPLETED" />
113:M >    <data
114:M >      android:scheme="package" />
115:M > </intent-filter>
     .
     .
     .
```

Listing 1 suggests that the four first requested permission could have been added automatically. Listing 2 present a duplicated component where suggests that the first was newly injected by a naive hacking script.

# 4 RESULTS AND DISCUSSION

In this section, we detail the findings of our study. We remind the reader that our main goal with this study is the detection of false negative in current mining sandbox approaches at a real-world and in-depth data set, and potential blindspots existing when detecting malware. Regarding blindspots, we hypothesize the presence of two major blindspots - trace taken by the app from the entry point to a sensitive API, and the differences in manifest file of repackaged apps. In Section **??**, we summarize the results of our study that estimates the performance of Droidbot test generation tools for mine Android Sandboxes. For this purpose, we present the amount of app pair that had divergent sensitive app sets. We also present the sensitive APIs generally more used at repackaged apps. We performance this initial study since it help solve our R1, and served as reference to solve R2 and R3.

The remainder of this Section is structured as follows. Section 4.1 presents the false negative rate of our experiment, and top 15 most used sensitive APIs injected by malicious apps present at our dataset, answering R1. Section 4.2 presents the results of our study analysing the impact of trace on sandbox approaches to detect malware thereby answering R2, and Section 4.3 presents the results of our study analysing the impact of modified manifest files on sandbox approaches to detect malware thereby answering R3. Section 4.4 presents some insights gained from the overall study and their potential implications.

## 4.1 False negative rate and Sensitive APIs more used

In this section, we describe the results of reproducing the state of the art Android sandbox approaches on the new dataset of 800 app pairs. We perform this experiment to ensure our analysis of the blind spots are done in the same playing field. Firstly, given a pair of apps, we first execute each version of app using test generation tool Droibot at DroidXP infrastructure for three minutes. We also repeat all this process three times.

After all executions, DroidXP produces a dataset with the sensitive methods that both app versions call at each execution. We consider that a test generation tool could construct a sandbox able to detect a specific malware, if a particular sensitive method is called only by the malicious version of the app. This check is done using a Python script that compares the set of sensitive methods, called at both app version (benign/malicious).

Hence, with DroidXP output we generate several reports based on union of each sensitive methods set, explored at each execution. In the end, It include a set of observations like set of sensitive methods access by each app version (benign/malicious), the sensitive methods access just for the malicious version (diff), and a summary that present at each rows: Test tool name used, app identification, and the number of different sensitive APIs find. This last information is crucial to calculate the false negative rate. If the set of sensitive methods access just by malicious app is empty, means that app is not a malicious, and therefore featuring a false negative.

With Droidbot, our results present that it could detect a total of *xx* sensitive APIs set different among 800 app pairs (xx.xx%) explored. The result of Droidbot diverge from previous works that benefited from the same tool, as Bao et al. (66.66%) [8], and Costa et al. (76.04%) [29]. However, at our work we also used the same app pair from these previous works, amount our 800 app pair dataset. When we check these specific sample from these works, we realized that we get percentages close to these previous works, (xx.xx%).

In the end, after explorer all sensitive APIs set used by all app pair, we calculated which are the sensitive APIs more used for our repackage apps. At our study we find that when we used Droidbot to explore each repackage app, it calls xx times sensitive APIs from our list of 162 sensitive APIs, calling at least xx sensitive APIs one time. We realized that the first 15 APIs more used accounts for more than half of all calls (xx%). We summarize our results at Table 1.

With these results, we believe that mine sandbox approach could have a relevant false negative rate, resulting in a potential amount of *blindspot*, which encouraged us to endorse efforts aimed at answering R2 and R3.

**Research Question: 1**

Our results present that we do not have a high diversity of sensitive APIs explored by malicious app. We find that among xxx call to sensitive APIs, more than half were carried out to 15 sensitive APIs (Table 1). Results also suggest that mine sandbox approach could have a high false negative rate when applied on repackaged versions of benign apps. This encourages the emergence of new proposals that can support mine sandbox revealing eventual *blindspot*.

## 4.2 Trace Analysis Results

In this section, we describe the results of our investigation on how the trace from the entry point to sensitive API could impact the accuracy of sandbox approaches in terms of Android malware detection. Initially we collect the call graphs of Droidbot using *Logcat*. We make a filter to collect just the traces between the app's entry point and calls to any sensitive method present at our previously list of 162 sensitive methods.

Finally, with callgraph from both app versions (benign/malicious), we spot differences between their traces. We choose to investigate only app pairs that were the same sensitive APIs detected at both version at first experiment, i.e, from the point of view of sandbox approaches, it sheds light on the blindspots. Our results show that among xx pair apps which had no difference between their sensitive APIs explored, xx of them (xx.xx%) had different trace from the entry point to sensitive methods. This result which gives us evidence that it is possible to improve malware detection if we also consider the trace between app's entry point and calls to sensitive APIs.

**Table 1: Sessitive APIs more used by repackage apps**

| Sensitive API | Occurrences | (%) |
|---|---|---|
| 01 <android.telephony.TelephonyManager: java.lang.String getDeviceId()> | 60 | 5.40 |
| 02 <android.net.wifi.WifiManager: android.net.wifi.WifiInfo getConnectionInfo()> | 50 | 4.50 |
| 03 <android.net.wifi.WifiInfo: java.lang.String getMacAddress()> | 49 | 4.41 |
| 04 <android.net.NetworkInfo: java.lang.String getExtraInfo()> | 49 | 4.41 |
| 05 <android.telephony.TelephonyManager: java.lang.String getSubscriberId()> | 47 | 4.23 |
| 06 <android.net.NetworkInfo: java.lang.String getTypeName()> | 43 | 3.87 |
| 07 <android.net.NetworkInfo: android.net.NetworkInfo State getState()> | 38 | 3.42 |
| 08 <android.database.sqlite.SQLiteOpenHelper: android.database.sqlite.SQLiteDatabase getWritableDatabase()> | 35 | 3.15 |
| 09 <android.database.sqlite.SQLiteDatabase: android.database.Cursor query(java.lang.String, ...,java.lang.String)> | 35 | 3.15 |
| 10 <android.telephony.TelephonyManager: android.telephony.CellLocation getCellLocation()> | 34 | 3.06 |
| 11 <android.database.sqlite.SQLiteOpenHelper: android.database.sqlite.SQLiteDatabase getReadableDatabase()> | 34 | 3.06 |
| 12 <android.telephony.gsm.GsmCellLocation: int getLac()> | 33 | 2.97 |
| 13 <android.telephony.gsm.GsmCellLocation: int getCid()> | 33 | 2.97 |
| 14 <android.telephony.TelephonyManager: java.lang.String getNetworkOperator()> | 27 | 2.43 |
| 15 <android.telephony.TelephonyManager: java.lang.String getLine1Number()> | 26 | 2,34 |

Table 2 summarizes the results of this investigation. The column **Same API set (SAPI)** shows the number of app pair which had the same sensitive APIs explored at the first experiment, i.e, considered an undetected malware. The column **Trace Different (TD)** shows the number of app pairs (among the undetected ones) that have different traces from the entry point to the sensitive method call. The column **Improv. %** shows how much the malware detection of each tool could have been potentially improved if we considered the trace (calculated using (1))

**Table 2: Summary of the results of trace analysis.**

| Tool | Same API set (SAPI) | Trace Different (TD) | Improv. (%) |
|---|---|---|---|
| Droidbot | 31 | 4 | 12.90 |

$$Improvement = \frac{traceDifferent(TD) \times 100}{Execution(NID)} \quad (1)$$

Figure 4 shows a example of trace injected at malicious version of the app **[com.android.remotecontrolppt]**. On that case, benign and malicious app version access the same sensitive method, *getSubscriberid()*. This sensitive method returns the mobile unique subscriber ID, and require manifest file permission "READ_PHONE_STATE", present in both app version. The original app accesses this methods through 2 traces (1 and 2), which suggest an aware action from app user. However, instead of the 2 original traces, the malicious version injected a third trace using as entry point a method which performs a stealth computation on a background thread, *doInBackground*, suggesting a action without user aware.FH: explain more about this example, and find another one


**Figure 4: Example of Malicious Trace.**

**Research Question: 2**

Test generation tools like Droidbot have a blind-spot when it comes to being aware of the trace taken from the entry point to a sensitive API call. Although Droidbot achieved better performance results when mining sensitives resources, it could have detected $xx\%$ of its undetectable app pairs had it considered trace as a factor.

## 4.3 Manifest File Analysis Results

In this section, we describe the results of our investigation on the impact of modified manifest files towards the accuracy of sandbox approaches. To this end, we check some particulars from Manifest file, that point to a likely suspicious behavior. In section 3.6, we illustrated that an automatic hacking script could inject duplicated permission and actions into the Manifest file. We looked out for such modifications in the malware

that went undetected by the test generation tools. We also check if among these apps, there were requests to new permissions, that were not initially requested by the benign version, or if there were excessive requests for permissions in the malicious version's manifest files. Table 3 summarizes our results.

**Table 3: Manifest File with duplicate code.**

| Tool | (NID) | (DP) | (DA) | (DP or DA) | (DP and DA) |
|---|---|---|---|---|---|
| Droidbot | 31 | 6 | 10 | 14 | 2 |

The column (SAPI) indicates the number of malware that went undetected during our first study (same as Table 2's Same API set (SAPI)). The second column (DP) indicates how many Manifest files had duplicated permission. Column (DA) denotes the number of malware with the duplicated actions in their manifest file.

A duplicate request to permission in a malicious version's manifest file should have been performed by a script. Droidbot could have detected *xx* of their undetectable malware (*xx*) had it considered duplicate permissions or actions in their detection strategy.

Finally, we investigate how many app pairs request a suspicious amount of permissions. Past works indicate that a benign Android app normally requests on average 4 permission, while a malicious apps version requests a median number of 7 permission[35][20]. To this end, we observe how many app pairs have more than a delta of 3 permission requests in comparison with the benign version. Table 4 presents the results.

**Table 4: Manifest File with suspicious amounts of permission requests.**

| Tool | (MP) | (DP) | (NDP) |
|---|---|---|---|
| Droidbot | 2 | 1 | 1 |

The second column (MP) present how many app pairs not detected by first experiment, have other suspicious number of permission requests (3 or more) in their manifest file. Third and fourth columns (DP)(NDP) presents how many app pairs out of these contain duplicated permission or not respectively. As we can see, just xx apps investigated had request a suspicious amount of permissions, xx with duplicated permission in your code, and other without.

**Research Question: 3**

We can conclude that sandbox approach also could had better accuracy if they considered the suspicious modifications at manifest file in their analysis.

## 4.4 Implications

The results presented so far have implications for both practitioners and researchers. We bring evidence that there are blindspots in Android sandbox approaches that if considered as a factor in malware identification, could improve mine sandbox technique. In a first step, we present that in our study, the accuracy of sandboxes approach was slightly lower than previous work. That is because our data set encompasses a wider range of malware, not considered at previous works. Our study points out that when use Droidbot test generator tool to explorer sensitive APIs at both versions, it was being able to identify *xx* sensitive dataset distinct (*xx.xx%*) in our dataset. We also show that among all sensitive APIs explored, xxxx was the API more injected at malicious app.

Next, we present a trace analysis at our data set which aims to investigate if there were some different traces between an entry point and call to any sensitive resource. We investigated traces, just from pair apps which we got different sensitive APIs set from first study. Our findings indicate that the trace analysis is also effective for malicious apps identification, and could support mine sandbox approach. Our First study had significant support of trace analysis when investigating the app pair with same sensitive APIs explored. Among *xxx* app pair investigated, we present that *xx* have different traces (%xx), featuring therefore as a suspicious app. The message from these findings is that exploring trace analysis in conjunction with mine sandbox approach is useful for researchers and practitioners to improve malicious app detection tasks.

Finally, the last study used static analysis at Android manifest file, to investigate if even naive techniques to construct malicious apps can go unnoticed for mine sandbox approach. In the previous section, Table 4 and 3 summarizes this investigation. Although seems to be a simple technique, we reported some suspicious manifest files from malicious apps which also was a blindspots in Android sandbox approaches.

For industry and academia, these results have implications. They reinforce that there are benefits of integrating auxiliary techniques to mine sandbox approach for malware identification. We also present evidence that its possible benefit from static analysis on suspicious Android manifest file for malware identification, even at malware with a low similarity coefficient regarding its benign version. The Venn-diagram of Figure XXX summarizes how Droidbot explorer, the trace analysis, and two kinds of suspicious manifest files can complement each other.

# 5 THREATS TO VALIDITY

# 6 CONCLUSIONS AND FUTURE WORK

# REFERENCES

[1] I. Comscore. Comscore. [Online]. Available: https://www.comscore.com/Insights/Presentations-and-Whitepapers/2018/Global-Digital-Future-in-Focus-2018

[2] W. J. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Trans. Software Eng.*, vol. 43, no. 9, pp. 817–847, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2630689

[3] "Statista," https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, accessed: 2022-02-10.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 259–269. [Online].

Available: https://doi.org/10.1145/2594291.2594299
[5] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2382–2400, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2948910
[6] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 37–48. [Online]. Available: https://doi.org/10.1145/2884781.2884782
[7] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE Computer Society, 2017, pp. 23–26. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.8
[8] L. Bao, T. B. Le, and D. Lo, "Mining sandboxes: Are we there yet?" in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 445–455. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330231
[9] F. H. da Costa, I. Medeiros, P. Costa, T. Menezes, M. Vinícius, R. Bonifácio, and E. D. Canedo, "Droidxp: A benchmark for supporting the research on mining android sandboxes," in *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 143–148. [Online]. Available: https://doi.org/10.1109/SCAM51674.2020.00021
[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short paper: a look at smartphone permission models," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011, pp. 63–68.
[11] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: a scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 71–82. [Online]. Available: https://doi.org/10.1145/2771783.2771795
[12] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2014, Austin, TX, USA, June 16-20, 2014*, S. Sanghavi, S. Shakkottai, M. Lelarge, and B. Schroeder, Eds. ACM, 2014, pp. 221–233. [Online]. Available: https://doi.org/10.1145/2591971.2592003
[13] B. Kim, K. Lim, S. Cho, and M. Park, "Romadroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app's manifest file," *IEEE Access*, vol. 7, pp. 72 182–72 196, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2920314
[14] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: multi-layer profiling of android applications," in *The 18th Annual International Conference on Mobile Computing and Networking, Mobicom'12, Istanbul, Turkey, August 22-26, 2012*, Ö. B. Akan, E. Ekici, L. Qiu, and A. C. Snoeren, Eds. ACM, 2012, pp. 137–148. [Online]. Available: https://doi.org/10.1145/2348543.2348563
[15] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu, "Droidmat: Android malware detection through manifest and API calls tracing," in *Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012, Kaohsiung, Taiwan, August 9-10, 2012*. IEEE Computer Society, 2012, pp. 62–69. [Online]. Available: https://doi.org/10.1109/AsiaJCIS.2012.18
[16] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android custom permissions demystified: From privilege escalation to design shortcomings," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 70–86. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00070
[17] H. Cai and B. G. Ryder, "A longitudinal study of application structure and behaviors in android," *IEEE Trans. Software Eng.*, vol. 47, no. 12, pp. 2934–2955, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2020.2975176
[18] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: towards obfuscation-resilient mobile application repackaging detection," in *7th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec'14, Oxford, United Kingdom, July 23-25, 2014*, G. Ács, A. P. Martin, I. Martinovic, C. Castelluccia, and P. Traynor, Eds. ACM, 2014, pp. 25–36. [Online]. Available: https://doi.org/10.1145/2627393.2627395
[19] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 217–228. [Online]. Available: https://doi.org/10.1145/2382196.2382222
[20] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. L. Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1269–1284, 2017. [Online]. Available: https://doi.org/10.1109/TIFS.2017.2656460
[21] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459. Springer, 2012, pp. 37–54. [Online]. Available: https://doi.org/10.1007/978-3-642-33167-1_3
[22] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, E. Bertino and R. S. Sandhu, Eds. ACM, 2012, pp. 317–326. [Online]. Available: https://doi.org/10.1145/2133601.2133640
[23] M. Maass, A. Sales, B. Chung, and J. Sunshine, "A systematic analysis of the science of sandboxing," *PeerJ Comput. Sci.*, vol. 2, p. e43, 2016. [Online]. Available: https://doi.org/10.7717/peerj-cs.43
[24] L. Bordoni, M. Conti, and R. Spolaor, "Mirage: Toward a stealthier and modular malware analysis sandbox for android," in *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds., vol. 10492. Springer, 2017, pp. 278–296. [Online]. Available: https://doi.org/10.1007/978-3-319-66402-6_17
[25] F. H. da Costa, I. Medeiros, T. Menezes, J. V. da Silva, I. L. da Silva, R. Bonifácio, K. Narasimhan, and M. Ribeiro, "Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification," *CoRR*, vol. abs/2109.06613, 2021. [Online]. Available: https://arxiv.org/abs/2109.06613
[26] K. Jamrozik and A. Zeller, "Droidmate: a robust and extensible test generator for android," in *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*. ACM, 2016, pp. 293–294. [Online]. Available: https://doi.org/10.1145/2897073.2897716
[27] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 318–329. [Online]. Available: https://doi.org/10.1145/2931037.2931044
[28] T.-D. B. Le, L. Bao, D. Lo, D. Gao, and L. Li, "Towards mining comprehensive android sandboxes," in *2018 23rd International conference on engineering of complex computer systems (ICECCS)*. IEEE, 2018, pp. 51–60.
[29] F. H. da Costa, I. Medeiros, T. Menezes, J. V. da Silva, I. L. da Silva, R. Bonifácio, K. Narasimhan, and M. Ribeiro, "Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification," *J. Syst. Softw.*, vol. 183, p. 111092, 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2021.111092
[30] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 95–109. [Online]. Available: https://doi.org/10.1109/SP.2012.16
[31] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Androzoo: collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 468–471. [Online]. Available: https://doi.org/10.1145/2901739.2903508
[32] H. Cai and B. Ryder, "Understanding application behaviours for android security: A systematic characterization," Department of Computer Science, Virginia Polytechnic Institute & State ..., Tech. Rep., 2016.
[33] "Logcat," https://http://developer.android.com/tools/help/logcat.html, accessed: 2020-03-15.
[34] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. L. Traon, "Improving privacy on android smartphones through in-vivo bytecode instrumentation," *CoRR*, vol. abs/1208.4536, 2012. [Online]. Available: http://arxiv.org/abs/1208.4536
[35] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. A. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Symposium On Usable Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012*, L. F. Cranor, Ed. ACM, 2012, p. 3. [Online]. Available: https://doi.org/10.1145/2335356.2335360