Mining Android Sandboxes Using Dynamic and Static Analysis: a Non-exact Replication Study

Francisco Handrick da Costa[a], Ismael Medeiros[a], Thales Menezes[a], João Victor da Silva[a], Ingrid Lorraine da Silva[a], Rodrigo Bonifácio[a], Krishna Narasimhan[b]

[a]*Computer Science Department, University of Brasília, Brazil*
[b]*Software Technology Group, TU Darmstadt, Germany*

Abstract

The use of sandboxes is an effective technique for malware analysis. However, although the use of dynamic analysis for mining Android sandboxes has been investigated before, little is known about the potential benefits of combining static and dynamic analysis for mining Android sandboxes. Accordingly, in this paper we present the results of two studies that investigate whether or not static analysis might complement and increase the performance of dynamic analysis tools for mining Android sandboxes. In the first study we conduct a non-exact replication of the Bao et al. work, a previous study that compares the performance of test case generation tools for mining Android sandboxes. Differently from the original work, here we isolate the effect of the static analysis tool (DroidFax) they used to instrument the Android apps in their experiments. This decision was motivated by the fact that DroidFax could have influenced the efficacy of the dynamic analyses positively—through the execution of specific static analysis algorithms. In our second study, we carried out a new experiment to investigate the efficacy of tainted analysis algorithms for mining Android sandboxes. To this end, we executed the FlowDroid tool to mine the source-sink flows from the benign/malign pairs of Android apps used in the Bao et al. study. Our study brings several findings. For instance, the first study reveals that DroidFax alone (static analysis) can detect 43.75% of the malwares in the dataset of the Bao et al. work, leading to an overestimation of the performance of the dynamic analysis tools. The results of the second study show that (static) tainted analysis is also practical for mining Android sandboxes, with a performance similar to that reached by dynamic analysis tools.

Keywords: Malware Detection, Mining Sandboxes, Android Platform, Static Analysis and Dynamic Analysis

1. Introduction

Almost two-thirds of the world population use mobile technologies [1], and the Android Operating System has dominated the market of smartphones, tablets, and others electronic devices [2]. Due to this growing popularity, the number of incidents related to Android malicious software (malware) has significantly increased [3, 4]. Security issues in Android software applications [1] have become a relevant research topic, and many techniques have been developed to identify vulnerabilities in Android apps [5], including the use of static analysis algorithms to identify privacy leaks, and to reveal the misuse of cryptographic primitives [6, 7], for instance. Another approach explored by researchers to mitigate the same problem is the use of dynamic analysis to mine Android sandboxes. In this proposal, given an app A, a sandbox S identifies sensitive resources access by A, during dynamic and static analysis, and then block future calls to sensitive resources unidentified during this exploratory phase, thereby protecting Android users from malicious behavior. [8].

The main idea of mining sandboxes is to explore the set of calls to sensitive API methods in a Android app, using different approaches for automatically testing Android apps. These analysis explore apps behavior and record all calls to sensitive APIs. A sandbox builds upon the calls to sensitive Android APIs, during the execution of the test cases (exploratory phase). The sandbox could then block future calls to other sensitive resources which diverge from those found in the exploratory phase. Using this approach, based on just test case generation tools, the sandbox accuracy depends on the exploratory capabilities of each testing tool used to mine the sandbox security rules.

Jamrozik et al. [8] argue in favor of dynamic analysis for mining sandboxes, instead of using static analysis. The authors claim that dynamic analysis exceeds static analysis on mining Android sandboxes because the analysis must often assume that additional behaviors are possible than actually would be. Otherwise, static analysis might determine that some behavior is impossible and therefore no longer needs to be checked at runtime. Thereby, code that is interpreted, decrypted, or downloaded at run time is out of scope for static analysis—even though these more *dynamic* strategies are often used to introduce malicious behavior. Nonetheless, the authors do not give any empirical validation about the limitations of static analysis to mine sandboxes. As a consequence, it is not clear whether and how both approaches (dynamic and static analysis) could complement each other in the process of mining Android sandboxes.

The same lack of understanding about the role of static analysis also appears in the work of Bao et al. [9] (hereafter BLL-Study), which present an empirical study that compares different dynamic analysis tools for testing and mining Android sandboxes. Their study leverages DroidFax [10] to instrument 102 pairs of Android apps (each pair comprising a benign and a malign version of an App) (B/M) and to collect the information needed to mine sandboxes (that is, the calls to sensitive APIs). Although the authors report a precision of at most 70% of dynamic analysis tools to differentiate the benign and malicious versions of the apps, the authors ignore the fact that DroidFax statically analyzes the Android apps and records calls to sensitive APIs (besides instrumenting the apps), which might lead to an over estimation of the performance of the dynamic analysis tools for mining sandboxes and introduce a possible threat to the conclusions of that

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

work.

Our goal is to understand how static analysis algorithms might complement dynamic analysis approaches in the process of mining Android sandboxes. To this end, we performed two studies. In the first, we conduct a replication of the work of BLL-Study. Our study differs from the original work in three aspects: (a) by isolating the effect of the static analysis component of DroidFax, (b) by considering more recent dynamic analysis tools for testing Android apps, and (c) we considering more execution time, and more repetitions, for each apps analysed at our replication. In the second study, we explore to what extent we can complement the dynamic analysis approach using tainted analysis algorithms, through Flowdroid [5] tool.

Extended Version. This journal paper extends a previous conference publication [11] by expanding the empirical evaluation of sandboxes solutions using DroidXP benchmark tool with:

(1) presenting a replication of the accuracy study of test generation tools for mining android sandboxes, using an extension and update of our benchmark solution, that aided us disables the static analysis performed by DroidFax.

(2) discovering which are the contribution installments of types of analysis for malware detection task.

(3) presenting what malware and its characteristics, that was not detected by any tool in our analysis dynamic and static experiment.

(4) presenting data that indicate that static analysis can improve the mining sandboxes process for malware detection.

The rest of the paper is organized as follows. Section 2 we present a background and some related works. Section 3 introduces the study setting of our research and Section 4 complements presenting some results and discussion, highlighting motivation malware examples and the use of a promising static analysis tool, on these examples. Finally, Sections 5 and 6, summarizes the paper, presenting some threats, presenting the final conclusions, and discussing our ideas for future works.

2. Background and related works

A sandbox is an isolated environment on an electronic device within which applications cannot affect other programs outside its boundaries, like the file system, the network or other device data. It enables testing and execution of unsafe or untested code, possible malware, without worrying about the integrity of the electronic device that runs the application.

The Android Mining Sandbox concept is a sandboxing technique that consists of mining rules from an android app, and use these rules to ensure system security. The sandboxing technique comprises of two steps. First, rules are miner and will compose the sandbox, through test generator tools. These tools explore program behavior and monitor access to sensitive APIs. The second stage ensures that resources not accessed, or accessed differently, at first stage, are denied access. So, if a malicious app requires access to resources, different from what was previously mined, the sandbox will prohibit this access.

Automatically mining software resources or components, to infer behavior is not new, and has been discussed before. For instance, Whaley et al. [12] combine dynamic and static analysis for API mining and so extract interface from software components. Ammons et al [13] propose a machine learning approach, called specification mining, to discover temporal and data-dependence relationships that a program follows when interacting with an API or abstract data type.

Regarding test generating tools used for mining Sandboxes, Jamrozik et al [14] proposed DroidMate, a test generation tool that implements a pseudo-random GUI exploration strategy, and was the first approach to leverage test generation to extract sandbox rules from apps. Li e tal. [15] proposed DroidBot, a test generator tool that explores sensitive resources access by apps, following a model-based exploration strategy. In their work, the authors present a comparison between DroidBot and Monkey [16] regarding malware analysis, and showed that Droidbot is able to trigger an amount of sensitive behaviors, like data leaks and file accesses, higher than Monkey. From the same authors, another test generator tool for Android, described as Humanoid [17], is a Droidbot evolution and presents a proposal that can generate humans like tests inputs, using deep learning.

L Bao et al .[9] had provided an important comparative study between five test generation tools that find malware by mining sandboxes techniques: DroidMate, Monkey, GUIRipper [18], Puma [19], and Droidbot. This study indicates that these tools was efficient in identifying at most 70% of the malware in a specific dataset and also reports that after combining all test generator tools, it was possible to detect 75.49% of malicious behavior explored (77 among 102). However, this study did not focus on the possible interference of static analysis in the final result, since this study used a instrumentation tool [10], that also performs a static analysis of the apps

Android applications contain within themselves the risk that sensitive data, such as credit card details, device information can be leaked into public sinks like the internet. Taint analysis is a special type of static analysis that enables tracking of sensitive data within programs [20]. Wei et al. [21] propose a scalable taint analysis for Android applications that applies traditional taint analysis techniques with targeted optimizations specific to the android operating system. Flowdroid [5] improves on precision of traditional approaches by including context and flow sensitivity. A significant issue with taint analysis is the cost of the tool itself hampering the performance. FastDroid [22] mitigates this issue by introducing an intermediate light-weight abstraction to perform the analysis.

3. Study Settings

BLL-Study explored the effectiveness of test generation tools to construct sandboxes, based on sensitive APIs called by apps under analysis. Since their study also involved a static analysis component, performed by Droidfax as we described at introduction, and whose impact was not measured, we replicated its work to understand the static analysis impact may have had in their results. Thus, this section present the settings of our study, whose goal is to perform this replication, and thus build a general understanding of the implications of static analysis algorithms in the results. We also investigate how static analysis can improve the performance of mining sandboxes, in the task of identifying malicious behavior.

To achieve these general goals, we intend to answer the following research questions.

(RQ1) What is the impact of the DroidFax static analysis algorithms in the BLL-Study?

(RQ2) What is the effective performance of each test generation tool, in terms of the number of detected malware, in BLL-Study, disregarding the DroidFax static analysis algorithms?

(RQ3) What are the benefits of using tainted analysis algorithms to complement the dynamic analysis provided by test generation tools for mining sandboxes?

Answering the research questions RQ1 and RQ2 allows us to disclose a possible overestimation performance of test generation tools for mining sandboxes, as reported in BLL-Study, which might introduce a possible threat to their conclusions. Answering the third research question allows us to open up the possibility of finding new strategies for designing mining sandbox techniques, complementing the performance of dynamic analysis through the use of static analysis algorithms.

We conducted two studies to answer the research questions above. First we replicated the BLL-Study. However, our study differs from the original because here we isolated the effect of the DroidFax static analysis algorithms, in the task to identify malicious apps. In addition, we discarded 6 pairs of Android apps we were not able to instrument—out of the 102 pairs used in the original work. We also introduced a recent test generator tool, that has not been considered at previous work, Humanoid [17], and different from the original study, we expand the execution time of each test generation tool, executing each app at each tool for three minutes, and repeated all the execution more 2 times, therefore totaling three executions. The original study executed each app at each tool, for just one minute, and just one time.

The others 3 test generation tools selected were: Droidbot [15], DroidMate [14] and Monkey [16], all explored at BLL-Study. We selected Droidbot and DroiMate because they achieved the best performance on detecting malicious behavior at 102 pairs of Android apps (B/M) in the BLL-Study. We also considered the open source tool from Google, Monkey, because it is the most widely used tool in industry [23]. It is part of the Android SDK, and does not require any additional installation effort. Since Monkey was developed for stress testing, and can not perform text input, only generating UI events, we also considered Humanoid, that emulate realistic users, creating human-like test inputs using deep learning techniques. For the replication, we also used our benchmark tool, DroidXP [11], which helped us to reproduce the original work.

In the second study we leverage FlowDroid [5] to execute tainted analysis algorithms in Android apps, in order to identify pairs of *sources and sinks*. In this case, our goal is to investigate the performance on detecting malicious behavior using static taint analysis of Android apps. We use two metrics in this second study: the number of source-sink flows that FlowDroid identifies when considering the pairs of Android apps (B/M), and the execution time of the analysis for each app. For both studies, we employed the same dataset 96 pairs of Android apps (B/M), shared by the AndroZoo [24] project. We detail the procedures of each study in what follows.

3.1. First Study: A replication BLL-Study

In the first study we executed the DroidXP benchmark with its default configuration, that is, enabling the DroidFax static analysis algorithms and the test generation tools.

We investigate the four test case generation tools described earlier and added a fake test case generation tool (named Joker) that simulates a test tool that does not execute the Android apps during a benchmark execution. Using this tool, the results of the dynamic analysis are not considered and we can compute the results with only the static analysis component of DroidFax (RQ1). Our study executed each pairs of Android app (B/M) in each one of the five test generation tools, including Joker, for three minutes, and for three times. We then investigate the performance of each tool to detect malicious behaviors in the sandbox modeled by the test generation tool under analysis. Figure 1 shows this experimental setup.

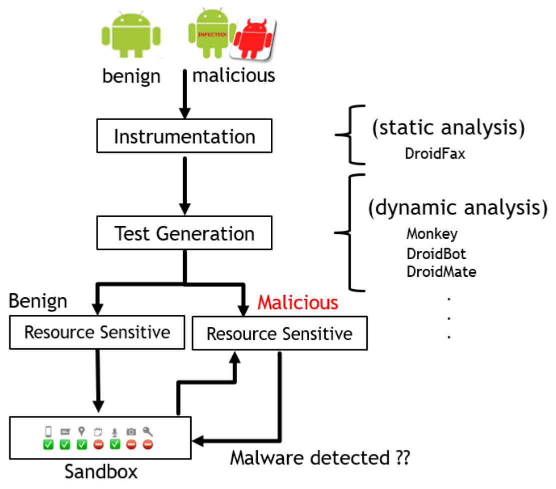


Figure 1: Experiment setup

After that, we replicated the study using a configuration of DroidXP that disables the DroidFax static analysis algorithm. With this new configuration, we executed the benchmark again, using the same dataset of Android pairs apps, the same execution time, and the same test case generation tools. The results of this replication allow us to compute the effective performance of the dynamic analysis tools (RQ2)—that is, ignoring the influence of the DroidFax static analysis algorithms.

3.2. Second Study: Tainted analysis algorithms.

To complement our initial study, we performed the tainted analysis at the same data set from the first study. For this second study, we used FlowDroid [25], a static flow analysis tool for Android apps. The goal is to test accuracy of static analysis, through tainted analysis algorithms, at finding malicious behavior, and compare the results with previous study at the same data set. In this second study, a behavior is considered malicious whenever the algorithms detected a different set of source-sink pairs, coming from the benign and malign versions of an Android app.

Initially, Flowdroid mine all sources and sinks of each benign apps, and enumerates all possible data flows between them. Next, it performs the same process for the malicious version of the correspondent app. In the final step, we compare the set of source-sink track between the benign app and its corresponding malicious app, in order to discovery some taint tracer different between the apps under analysis.

This study was useful to answer the third research question (RQ3), because it disclosed some pairs of applications with malicious behavior, that could not be detected at first study at none of the scenarios presented, evidencing that there is a benefit of new static analysis techniques, to complement the dynamic analysis provided by test generation tools, at mining sandboxes solutions. Details of studies results are at next section.

4. Results and discussion

In this section we detail the findings of our study. We present the results of the first and second studies in Section 4.1 and Section 4.2. In Section 4.3 we summarize the implications of our study.

4.1. Result of the first study: A non-exact replication

Our first study is a replication of the BLL-Study. As discussed in the previous section, to this end, we first executed the analysis using the DroidXP benchmark with its default configuration. After that, we executed the analysis again, though disabling the DroidFax static analysis algorithm. In this way, we could better estimate the performance of the dynamic analysis tools for mining Android sandboxes. Table 1 summarizes the results of the executions. The columns Exec. (WS) and Exec. (WOS) show the number of malwares identified when executing each tool with the support of the DroidFax static analysis algorithms (WS) and without the support of DroidFax static analysis algorithms (WOS). The Impact column shows the impact (in percentage) of DroidFax static analysis algorithms in the results.

Note taht, in the BLL-Study, the authors do not present a discussion about the influence of DroidFax in the results, even though here we report that the impact of DroidFax in the results is significant, ranging from 16.44% (DroidBot) to 51.79% (Humanoid)—discarding our Joker tool for which DroidFax improves its performance by 100%. This result is expected, since the Joker tool does not execute any dynamic analysis. Next we discuss the result of each individual test generation tool.

Table 1: Summary of the results of the first study.

Tool	Exec. (WS)	Exec. (WOS)	Impact (%)
DroidBot	73	61	16.44
Monkey	71	56	21.13
DroidMate	68	52	23.53
Humanoid	56	27	51.79
Joker	42	0	100.00

DroidBot in the first execution (Exec. WS) led to a sandbox that detected a total of 73 malware among 96 pairs present in our dataset (76.04%), detecting more apps with malicious behavior than any other tool. Similar to the BLL-Study, DroidBot is the test case generation tool whose resulting sandbox detected the largest number of malicious apps. Moreover, in our second execution (Exec WOS), removing the DroidFax static analysis support reduced the DroidBot performance in 16.44%, the smaller impact we observed among the tools.

Monkey in the first execution (Exec. WS) produced a sandbox that detected 71 out of the 96 pairs of Android apps. Contrasting, in the original study, the Monkey's sandbox detected 48 malwares within the 102 pairs (47.05%). This difference might be due to the fact that Monkey uses a random strategy for test case generation and here we considered the outcomes of three executions—while in the BLL-Study, the authors consider the outcomes of one execution. Considering our second execution (Exec WOS), there is a reduction of 21.13% in the Monkey's performance, leading to a sandobox that was able to detect 56 malwares.

DroidMate in the first execution (Exec. WS) led to a sandbox that detected 68 apps with malicious behavior (70.83%). In the BLL-Study study, DroidMate also detected 68 malwares, though considering the 102 pairs of apps used in the original study. In the second execution(WOS), without the DroidFax static analysis algorithms, the resulting sandbox's performance drops by 23.53%, being able to detect 52 out of the 96 pairs of Android apps.

Humanoid showed the worse performance, even though a previous work [17] presented that it leads to the highest number of lines coverage in comparison to Monkey, DroidBot, and DroidMate. In the first execution (Exec. WS), the resulting Humanoid sandbox identified 56 malwares in our dataset

(58.33%). Humanoid was the most affected in the second execution (Exec. WOS), whose resulting sandbox presents a redunction of 51.79% in the number of detected malwares. Since the BLL-Study did not explore Humanoid, we do not have a baseline for comparison with the previous work.

Joker is our fake test case generation tool that help us understand the performance of the DroidFax static analysis algorithm for mining sandboxes. We integrated Joker into the DroidXP benchmark as an additional test case generation tool that does not run the Android apps during the benchmark execution. As a result, the analysis using Joker reveals the performance of DroidFax static analysis algorithms alone. For the first execution, with the DroidFax static algorithms enabled, even though Joker does not execute the Android apps, its resulting sandbox detected 43.75% of the malwares. For the second execution, that is, disabling the DroidFax static analysis algorithm, the resulting Joker sandbox was not able to detect any malware. This result was expected, since Joker does not analyze the Android apps during the benchmark execution.

Finding 1. *Integrating the dynamic analysis tools with the DroidFax static analysis algorithms improves substantially the performance of the resulting Android sandboxes for detecting malicious behavior.*

The Venn-diagram of Figure 2 summarizes how the tools can complement each other. Note in the diagram that 53 malwares have been detected by all sandboxes generated in the first execution (with the DroidFax static analysis algorithms), out of the 78 identified by at least one sandbox. In addition, the DroidBot sandbox did not detect any malware that had not been detected by the other tools. Differently, the Monkey sandbox detected three malwares that had not been detected by any other sandbox, the DroidBot sandbox detected two malwares that had not been detected by any other sandbox, and the Humanoid sandbox detected one malware that had not been detected by any other sandbox. Contrasting with the BLL-Study, our results suggest that using DroidBot in combination with Monkey, DroidMate, and Humanoid does not improve the general performance of an integrated environment for mining Android sandboxes.

Finding 2. *Our results suggest that one might benefit from using an integrated environment that combines Monkey, DroidMate, and Humanoid to mine Android sandboxes. Introducing the DroidBot tool does not improve the results.*

Altogether, ignoring the Joker tool, our study reveals that from 58.33% (Humanoid) to 76.04% (DroidBot) of the malicious apps investigated in our study can be detected using the sandboxes generated after running the test case tools with the support of the DroidFax static analysis algorithms. Besides that, in the first execution, none of the resulting sandboxes could detect 18 malwares in our dataset (18.75%). According to the Euphony tool [26], 12 of these 18 malwares are *adwares*, 3 are *trojans*, 2 are PUPs (*Potentially Unwanted Program*), and one is an *exploit*.

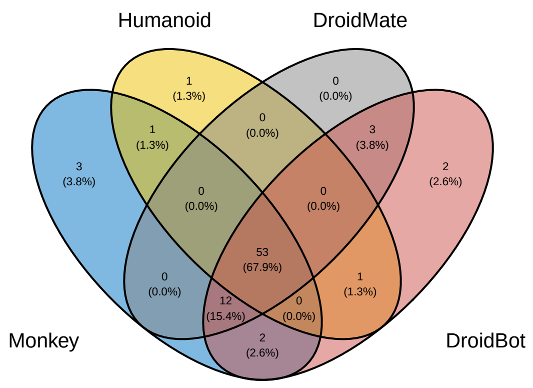


Figure 2: Venn Diagram highlighting how the sandboxes from the tools can complement each other.

The DroidBot and Humanoid sandboxes detected the malicious version of the app `com.andoop.flyracing`. In this particular case, the malicious version changed the Android Manifest file, adding permissions to receive and send SMS messages (Listing **??**). Moreover, after decompiling this malware, we also observed that the malicious version of the `MainService` class introduced a behavior that collects sensitive information (the International Mobile Equipment Identity, IMEI) and sends it using an SMS message (Listing **??**).

Listing 1: Diffs in the `com.gau.screenguru.finger` AndroidManifest file of the malicious version

```
67:M >    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
68:M >    <uses-permission android:name="android.permission.SEND_SMS"/>
```

Listing 2: Diffs in the malicious version of the class `com.android.main.MainService` (app `com.gau.screenguru.finger`)

```
492:M > localObject2 = (TelephonyManager)getSystemService("phone");
493:M > if (localObject2 != null)
494:M > {
495:M >  this.imei = ((TelephonyManager)localObject2).getDeviceId();
496:M >  this.imsi = ((TelephonyManager)localObject2).getSubscriberId();
497:M >  this.iccid = ((TelephonyManager)localObject2).getSimSerialNumber();
498:M > }
// [...]
519:M > if ("".equals(this.destMobile)) {
520:M >  getDestMobile();
521:M > }
522:M > sendSMS(this.destMobile, "imei:" + this.imei)
```

Differently, none of the computed sandboxes detected the malicious version of the app `com.andoop.flyracing`. Indeed, the malicious version only changes the Android Manifest file, modifying the meta-data `ADMOB_PUBLISHER_ID`. However, the AdMob is a monetization service provided by Google, and changing the AdMob *publisher identifier* account redirects the advertisement's revenue to another destination.

Listing 3: Diff in the file `com.andoop.flyracing` AndroidManifest file of the malicious version. `B` stands for the benign version, while `M` stands for the malign version.

```
1:B < <meta-data android:name="ADMOB_\PUBLISHER_\ID"
                    android:value="a14cf7346295891"/>
---
1:M > <meta-data android:name="ADMOB_\PUBLISHER_\ID"
                    android:value="a14f099bfbf3c61"/>
```

4.2. Result of the second study: Tainted analysis algorithms.

In this second study we used a tainted analysis approach to mine differences between the benign and malicious versions of each of the 96 Android apps in our dataset. To this end we leverage the FlowDroid tool, which tracks how sensitive information flows through the apps using tainted analysis algorithms. Regarding accuracy, the tainted analysis approach detected 58 out of the 96 pairs in our dataset $(60,42\%)$, that is, the FlowDroid approach leads to a better performance than any sandbox originated in the second execution of the dynamic analysis tools (without the DroidFax static analysis algorithms).

Finding 3. *The performance of FlowDroid to identify malicious behavior is superior than the performance of the mining sandbox approach supported by dynamic analysis only—without the DroidFax static analysis algorithms.*

Additionally, we investigate if we could benefit from combining the results from FlowDroid and DroidFax. Figure 3 shows a Venn-diagram summarizing the results. So, when combining the results from FlowDroid and DroidFax, we were able to detect 67 of the malicious apps (69.79%), a performance compatible to that we found as response to the first execution of the test case generation tools—which also considers the DroidFax static analysis algorithms. More interesting, from those 67 malicious apps identified, 33 pairs had been found by both tools (FlowDroid and DroidFax), even though they follow a completely different static analysis approach. Furthermore, FlowDroid shows to be more effective than DroidFax, detecting 25 malicious apps that had not been detected by DroidFax (while DroidFax detected 9 malicious apps that had not been detected by FlowDroid).

Finding 4. *Integrating the results of static analysis tools (such as FlowDroid and DroidFax) seems promising, leading to a performance similar to that achievend when combining test case generation tools with the DroidFax tool.*

The execution of FlowDroid is also feasible: the analysis takes only 32.08 seconds per app on average, totaling a processing time of 52 minutes to analyze all 96 pairs of Android apps. Even though the time to execute the FlowDroid analysis depends on the size of the app, the longest run took only 437 seconds.

Finally, we can highlight that FlowDroid was able to detect 4 malwares among the 18 malicious Android apps that had not been detected by the sandboxes constructed in the first study. Among these four malwares, 2 are *trojans*, 1 is an *exploit*, and 1 is an *adware*.

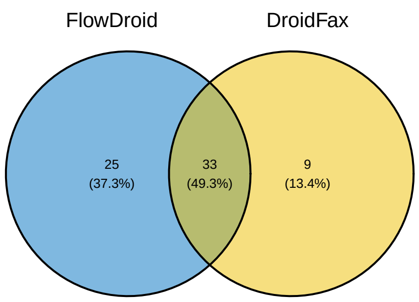
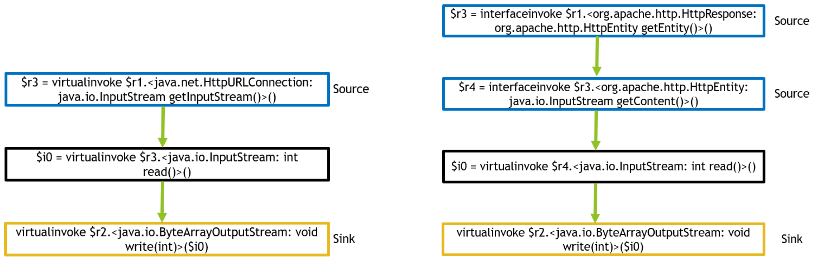


Figure 3: Venn Diagram highlighting the possible benefits of integrating FlowDroid and DroidFax.

Finding 5. *Although FlowDroid presents a performance similar to that of using the dynamic analysis approach for mining sandboxes, it was able to detect only four additional malwares (out of the 18) that had not been detected in the first study.*

Here we present details about one of the malware that Flowdroid had detected. At this package (com.yy.fontmaster), from another alternative android market, angeeks [?], the malicious and benign apps have the same sink, however they access different sources, therefore configuring additional source-sink pairs, and hence a possible malicious behaviours. The Figure 4, presents the paths source (blue border) and sink (orange border) from benign and malicious apps.



```
$r3 = virtualinvoke $r1.<java.net.HttpURLConnection: java.io.InputStream getInputStream()>()
$i0 = virtualinvoke $r3.<java.io.InputStream: int read()>()
virtualinvoke $r2.<java.io.ByteArrayOutputStream: void write(int)>($i0)
```

```
$r3 = interfaceinvoke $r1.<org.apache.http.HttpResponse: org.apache.http.HttpEntity getEntity()>()
$r4 = interfaceinvoke $r3.<org.apache.http.HttpEntity: java.io.InputStream getContent()>()
$i0 = virtualinvoke $r4.<java.io.InputStream: int read()>()
virtualinvoke $r2.<java.io.ByteArrayOutputStream: void write(int)>($i0)
```

(a) Path source/sink from benign app (b) Path source/sink from malicious app

Figure 4: Pairs source/sink different (B/M)

4.3. Discussion

First we find that static analysis summaries had impact in the BLL-Study. It was responsible for improving the results of the tools, between 16.44% and 51.79%, discarding our Joker tool, answering our first research question (RQ1). Second, Table 1 also summarizes our findings addressing the second research question (RQ2). We realized that disregarding the static analysis algorithms, and discarding Joker tool, among the tools analyzed in your study, Humanoid had the biggest performance drop, obtaining an effective performance of 28.12%, 27 malicious apps, among 96. The least affected was DroidBot, proving be the tool with better effective performance, in terms of the number of detected malware, obtaining an effective performance of 63.54%, 61 malicious apps, among 96 . Finally, we answered our last research question (RQ3) when we leverage sandboxes, complementing the dynamic analysis provided by test generation tool with tainted analysis algorithms. Our experiment highlight that 69.79%-82.29% of malware in dataset can be now uncovered by the complement of tainted analysis algorithms, evidencing that sandboxes can be further boosted when coupled with new static analysis techniques. We found that the number of identified malicious apps detected is increased for all cases, achieving at the best case, 82.29% with DroidBot, a better performance than all five tools explored at BLL-Study, even when they constructed a sandbox by combining all test generation tools at theirs work 75.49%. Table 2 display the increase of all tools explored at our research.

Test Generation Tool	FlowDroid Increase	Total	%
DroidBot	6	79	82.29
Monkey	7	78	81.25
DroidMate	7	75	78.12
Humanoid	16	72	75.00
Joker	25	67	69.79

Table 2: Malwares detected in 96 pair (B/M) increased by Tainted analysis Algorithms

5. Threats

As any empirical work, this work also has limitations and threats to its validity.

Internal Validity. Due to the randomization behavior presented by each chosen tool, we should not validate the results of this experiment without considering the presence of random events in the execution. To mitigate this, we have used a configuration of the benchmark DroidXP tool that runs multiple times each test and computes the average result from those executions. So, the comparison between the results of this experiment and the experiment presented by Bao et al. could be more precise. Beyond that, we tested only 96 of the original 102 pairs of apps in this experiment because the benchmark could not execute those six pairs of apps due to crashes in the Android emulator. This difference threatens the validity of the comparison between the Bao et al.'s experiment and ours.

External Validity. As this is a replication study, it brings some of the same threats BLL-Study had presented. This study used the same set of apps, which were from a piggy-backed app dataset released by Li et al. [29] So, using this dataset, we could not cover all categories of Android malware. Besides that, we only used a small number of test case generation tools in this study. To mitigate these threats and enrich our research data, we plan to test more apps and use other test case generation tools.

In this study, we used a single tool for Tainted Analysis. With such a small number of tools, we could not have a sample of what a static analysis tool could help in Tainted Analysis. So, we plan to use more tools to mitigate this threat.

6. conclusion and future work

Mining sandboxes is a popular technique to isolate android applications to analyse their behavior and discover vulnerabilities. Currently most sandbox mining approaches use dynamic analysis and have been studied previously in detail by many including BLL-Study. But the impact of static analysis on these mining approaches is unexplored currently. The use of static analysis techniques, like taint analysis algorithms, to detect malicious behaviours, seems promising and can leverage sandboxes accuracy. However, many authors argue in favor of dynamic analysis for mining sandboxes, highlight limitations at static analysis at this task, neglecting the complement that could be obtained with it. To the best of our knowledge, this is the first work that explored the impact of static analysis at process of mining Android sandboxes for malware detect.

In this paper, we performed two studies. In the first we conducted a non-exact replication of BLL-Study, considering 96 pairs apps (B/M) from the original study, and selecting 5 test generation tools to run for 1 minute for 2 times. Firstly, we performed the study, using DroidXP benchmark with its default configuration, enabling the Droidfax static analysis algorithms, and repeat the study by disabling Droidfax, to compute the effective performance of the test generation tools, ignoring the influence of static analysis. For the second experiment, we complement the dynamic analysis using tainted analysis algorithms, through Flowdroid tool, over the same 96 pairs (B/M) of the previous experiment. The results of the first experiment reveals that, ignoring our fake test case generation tool (Joker), that does not executed the apps, 51.04%-73.95% of the malware investigated was detected using the sandboxes with the support of Droidfax static analysis algorithms. Enabling the Droidfax, the sandoxes improved its performances from 23.94% (Monkey) to 69.39% (Humanoid). We also find that 19.79% of malicious apps (19 out of 96) could be detected by none of the sandboxes generated. In the second experiment, the tainted analysis detected 58 among 96 pairs investigated (60.42%), a better performance than any sandbox constructed by the execution of the test case tools without the Droidfax static analysis algorithms.

As future work, we plan to investigate other promising tools to enhance our data from previous executions. We will use the benchmark to test tools like Sapienz (a search-based test generation tool) [30] and Dynodroid (an input generation system for Android apps) [31]. And we will search for other great tools used by academia and by the industry to be tested by the benchmark DroidXP. Also, we will execute each of the tools used in this experiment without the benchmark. And, we will compare these results with this experiment's results.

This comparison will be valuable in the pursuit of one of this study's goals, which was to analyze what impact the benchmark droidXP has on its results.

References

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

[2] statcounter, https://gs.statcounter.com/os-market-share/mobile/worldwide, accessed: 2021-02-10.

[3] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, M. Rajarajan, Android security: A survey of issues, malware penetration, and defenses, IEEE Commun. Surv. Tutorials 17 (2) (2015) 998–1022. doi:10.1109/COMST.2014.2386139.
URL https://doi.org/10.1109/COMST.2014.2386139

[4] Sufatrio, D. J. J. Tan, T. Chua, V. L. L. Thing, Securing android: A survey, taxonomy, and challenges, ACM Comput. Surv. 47 (4) (2015) 58:1–58:45. doi:10.1145/2733306.
URL https://doi.org/10.1145/2733306

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, P. D. McDaniel, Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, in: M. F. P. O'Boyle, K. Pingali (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, ACM, 2014, pp. 259–269. doi:10.1145/2594291.2594299.
URL https://doi.org/10.1145/2594291.2594299

[6] S. Krüger, J. Späth, K. Ali, E. Bodden, M. Mezini, Crysl: An extensible approach to validating the correct usage of cryptographic apis, in: T. D. Millstein (Ed.), 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands, Vol. 109 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 10:1–10:27. doi:10.4230/LIPIcs.ECOOP.2018.10.
URL https://doi.org/10.4230/LIPIcs.ECOOP.2018.10

[7] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, D. D. Yao, Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 2455–2472.

doi:10.1145/3319535.3345659.
URL https://doi.org/10.1145/3319535.3345659

[8] K. Jamrozik, P. von Styp-Rekowsky, A. Zeller, Mining sandboxes, in: L. K. Dillon, W. Visser, L. A. Williams (Eds.), Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, 2016, pp. 37–48. doi:10.1145/2884781.2884782.
URL https://doi.org/10.1145/2884781.2884782

[9] L. Bao, T. B. Le, D. Lo, Mining sandboxes: Are we there yet?, in: R. Oliveto, M. D. Penta, D. C. Shepherd (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, IEEE Computer Society, 2018, pp. 445–455. doi:10.1109/SANER.2018.8330231.
URL https://doi.org/10.1109/SANER.2018.8330231

[10] H. Cai, B. G. Ryder, Droidfax: A toolkit for systematic characterization of android applications, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017, IEEE Computer Society, 2017, pp. 643–647. doi:10.1109/ICSME.2017.35.
URL https://doi.org/10.1109/ICSME.2017.35

[11] F. H. da Costa, I. Medeiros, P. Costa, T. Menezes, M. Vinícius, R. Bonifácio, 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. doi:10.1109/SCAM51674.2020.00021.
URL https://doi.org/10.1109/SCAM51674.2020.00021

[12] J. Whaley, M. C. Martin, M. S. Lam, Automatic extraction of object-oriented component interfaces, in: P. G. Frankl (Ed.), Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002, ACM, 2002, pp. 218–228. doi:10.1145/566172.566212.
URL https://doi.org/10.1145/566172.566212

[13] G. Ammons, R. Bodík, J. R. Larus, Mining specifications, in: J. Launchbury, J. C. Mitchell (Eds.), Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002, ACM, 2002, pp. 4–16. doi:10.1145/503272.503275.
URL https://doi.org/10.1145/503272.503275

[14] K. Jamrozik, 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. doi:10.1145/2897073.2897716.
URL https://doi.org/10.1145/2897073.2897716

[15] Y. Li, Z. Yang, Y. Guo, X. Chen, Droidbot: a lightweight ui-guided test input generator for android, in: S. Uchitel, A. Orso, M. P. Robillard (Eds.), Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, IEEE Computer Society, 2017, pp. 23–26. doi:10.1109/ICSE-C.2017.8.
URL https://doi.org/10.1109/ICSE-C.2017.8

[16] Monkey, https://developer.android.com/studio/test/monkey, accessed: 2020-02-10.

[17] Y. Li, Z. Yang, Y. Guo, X. Chen, Humanoid: A deep learning-based approach to automated black-box android app testing, in: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, IEEE, 2019, pp. 1070–1073. doi:10.1109/ASE.2019.00104.
URL https://doi.org/10.1109/ASE.2019.00104

[18] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, A. M. Memon, Using GUI ripping for automated testing of android applications, in: M. Goedicke, T. Menzies, M. Saeki (Eds.), IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, ACM, 2012, pp. 258–261. doi:10.1145/2351676.2351717.
URL https://doi.org/10.1145/2351676.2351717

[19] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, R. Govindan, PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps, in: A. T. Campbell, D. Kotz, L. P. Cox, Z. M. Mao (Eds.), The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014, ACM, 2014, pp. 204–217. doi:10.1145/2594368.2594390.
URL https://doi.org/10.1145/2594368.2594390

[20] F. Pauck, E. Bodden, H. Wehrheim, Do android taint analysis tools keep their promises?, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 331–341. doi:10.1145/3236024.3236029.
URL https://doi.org/10.1145/3236024.3236029

[21] W. Huang, Y. Dong, A. Milanova, J. Dolby, Scalable and precise taint analysis for android, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Association for Computing Machinery, New York, NY, USA, 2015, p. 106–117. doi:10.1145/2771783.2771803.
URL https://doi.org/10.1145/2771783.2771803

[22] J. Zhang, C. Tian, Z. Duan, An efficient approach for taint analysis of android applications, Computers and Security 104 (2021) 102161. `doi:https://doi.org/10.1016/j.cose.2020.102161`.
URL `https://www.sciencedirect.com/science/article/pii/S016740482030434X`

[23] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, T. Xie, Automated test input generation for android: are we really there yet in an industrial case?, in: T. Zimmermann, J. Cleland-Huang, Z. Su (Eds.), Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, ACM, 2016, pp. 987–992. `doi:10.1145/2950290.2983958`.
URL `https://doi.org/10.1145/2950290.2983958`

[24] K. Allix, T. F. Bissyandé, J. Klein, Y. L. Traon, Androzoo: collecting millions of android apps for the research community, in: M. Kim, R. Robbes, C. Bird (Eds.), Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016, ACM, 2016, pp. 468–471. `doi:10.1145/2901739.2903508`.
URL `https://doi.org/10.1145/2901739.2903508`

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, SIGPLAN Not. 49 (6) (2014) 259–269. `doi:10.1145/2666356.2594299`.
URL `https://doi.org/10.1145/2666356.2594299`

[26] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, L. Cavallaro, Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 425–435.

[27] admob, `https://admob.google.com/home/`, accessed: 2021-02-10.

[28] publisherid, `https://support.google.com/admob/answer/2784578?hl=en`, accessed: 2021-02-10.

[29] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, L. Cavallaro, Understanding android app piggybacking: A systematic study of malicious code grafting, IEEE Transactions on Information Forensics and Security 12 (6) (2017) 1269–1284. `doi:10.1109/TIFS.2017.2656460`.

[30] K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for android applications, in: A. Zeller, A. Roychoudhury (Eds.), Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, ACM, 2016, pp. 94–105. `doi:10.1145/2931037.2931054`.
URL `https://doi.org/10.1145/2931037.2931054`

[31] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: an input generation system for android apps, in: B. Meyer, L. Baresi, M. Mezini (Eds.), Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, ACM, 2013, pp. 224–234. doi:10.1145/2491411.2491450.
URL https://doi.org/10.1145/2491411.2491450