The Achilles' Heel of the Android Mining Sandbox Approach for Malware Identification

Anonymous
*Institution

***Abstract*—Android is the most popular operating system for the mobile platform, and smartphones' ubiquitous nature in our daily lives has only made their security an important topic for researchers and practitioners alike. Previous research results have advocated using the Mining Android Sandbox Approach (MAS approach) to identify malicious behavior in repackaged apps, one of the most popular methods to inject malicious behavior into android apps. Nonetheless, these previous studies have drawn their conclusions using a small dataset of 102 pairs of original and repackaged apps, threatening the findings w.r.t. external validity and opening the question of whether or not the MAS approach scales to larger and more diverse datasets. To mitigate these issues, we conduct a new experiment that reproduces the state-of-the-art research that empirically evaluated the MAS approach performance. Our reproduction study uses a dataset that is an order of magnitude larger than the datasets used in previous research (a total of 1203 pairs of apps with a much diverse distribution of malware families, for instance). To our surprise, our experiments revealed that the accuracy rate of the MAS approach for malware identification drops significantly: F_1 score drops from 0.89 in the previous dataset to 0.42 in our larger dataset. After an in-depth assessment, we found that the representative number of malware from the gappusin family explains the higher number of samples for which the MAS approach fails to correctly classify as malware. Our findings open the discussion on the possible blindspots that plague the MAS approach and their accuracy issues when scaled and reveal the need for complementing the MAS approach with other techniques so that it could effectively detect a broader class of malware.**

***Index Terms*—Android Malware Detection, Dynamic Analysis, Mining Android Sandboxes**

I. Introduction

Mobile technologies like smartphones and tablets have become fundamental to the way we function as a society. Almost two-thirds of the world population uses mobile technologies [1], [2], with the Android Platform dominating this market accounting for more than 70% of the *mobile market share* with almost 3.5 million Android applications [1] (apps) available on the Google Play Store [3]. With increased popularity also comes increased risk of attacks—motivating plenty of research efforts to design and develop new techniques to identify malicious behavior or vulnerable code in Android apps [4].

For instance, the mining sandbox approach takes advantage of automated test case generation tools to explore the behavior of an app—in terms of calls to sensitive APIs—and then generates an Android sandbox [5]. During a normal execution of the app, the sandbox might block any call to a sensitive API that had not been observed during the exploratory phase. Researchers have shown that the mining sandbox approach is also effective in detecting a popular class of Android malware that "repackages" benign apps [6], [7]—i.e., starting with a benign version of an app from an official app store, such as Google Play, one might infect it with malicious code, such as broadcasting sensitive information to a private server [8]. The infected app is then shared with users using different app stores. Previous research works [6], [9] compare the accuracy of Android sandboxes for malware detection produced from the execution of different test case generation tools, including Monkey [10], Droidbot [11], and Droidmate [12] tools. These reports bring evidence that the test generation tool Droidbot outperforms the other tools, leading to sandboxes that classify as malware 70% of the repackaged apps in a dataset.

Although revealing promising results, the previous research has two main limitations. First, they use a small dataset of malware comprising only 102 pairs of benign/repackaged versions of an app. Second, their impact analysis is not fine-grained. For instance, they do not consider whether the impact on the accuracy that might be due to malware characteristics, such as the similarity index between the benign and the repackaged version of an app and the malware family (gappusin, kuguo, dowgin, etc.). So, in this paper we present the results of an investigation that aims to replicate previous studies [6], [13] in a larger dataset of app pairs (benign/repackaged versions), and then explore whether a more diverse sample of app pairs has an influence on the accuracy of the mining sandbox approach for malware identification. To this end, we explore the performance of the mining sandbox approach using a larger dataset we curated for this research and Droidbot [11] as test case generation—the tool that, according to the literature, leads to the most accuracy Android sandbox. Our new dataset is an order of magnitude larger (it contains 1203 pairs of benign/malicious apps), comprises a much more diverse similarity index, and covers a broader range of malware families.

Negative results. To our surprise, the experimental results show that, on a larger and more diverse dataset (compared to previous studies), the mining sandbox approach's accuracy (F_1 score) drops significantly (from 0.89 to 0.42). This result motivated us to conduct a series of experiments to understand the reasons for the lower accuracy in the larger dataset and also explore an extension of the original mining sandbox approach for malware identification. The original approach classifies an app as malware whenever there exists a difference between

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

the sets of calls to sensitive APIs collected when running the test case generation tool over the original and repackaged versions of the same app. In our extension, we also classify as malware a version of an app that calls the same set of sensitive APIs collected in the exploratory phase but with at least one different trace from the app's entry points to the collected sensitive method calls.

Our experiments show that the number of **false negatives** reduces when they are made aware of the differences in the dynamic call trace, improving the recall 9% in our large dataset (from 37% to 46%). Nonetheless, even using the trace analysis extension, the MAS approach fail to correct classify the samples from the *gappusin* malware family (a particular class of adware that frequently appears in repackaged apps). Out of the total of 198 samples within this family, the vanilla MAS approach and its trace-based variant fail to correctly classify 171 samples as malware. Our results reveal that this particular family is responsible for the low accuracy ($F_1 = 0.42$) of the MAS approach in the large dataset. An accuracy of 42% may still appear unsatisfactory for a truly trustworthy sandbox approach. Still, our experiments open the discussion (a) on one important blindspot that must be considered when building malware detection approaches that mine sandboxes, and (b) on the need to investigate further sources of information when trying to distinguish benign and malign app versions.

We detail the MAS approach and the MAS approach for malware detection in Section II. Next, in Section III, we characterize our experiments in terms of research goal, questions, metrics, datasets, and procedures for data collection and data analysis. Section IV and Section V detail the main findings of our research and possible decisions that might threat the validity of our results. Finally, Section VI presents final remarks and possible future work. The main assets we produced during this research are available in the paper repository.[2]

II. Background and Related Work

There are many tools that favor developers to reverse engineering the Android bytecode language [14]. For this reason, software developers can easily decompile trustworthy apps, modify their contents by inserting malicious code, repackage them with malicious payloads, and re-publish them in app stores, including official ones like the Google Play Store. It is well-known that repackaged Android apps can leverage the popularity of real apps to increase its propagation and spread malware. Repackaging has been raised as a noteworthy security concern in Android ecosystem by stakeholders in the app development industry and researchers. Indeed, there are reports claiming that about 25% of Google Play Store app content correspond to repackaged apps [15]. Nevertheless, all the workload to detect and remove malware from markets by the stores (official and non-official ones), have not been accurate enough to address the problem. As a result, repackaged Android apps threaten security and privacy of unsuspicious Android app users, beyond compromising the copyright of the original developers [16]. Aiming at mitigating the threat of malicious code injection in repackaged apps, several techniques based on both static and dynamic analysis of Android apps have been proposed.

A. Mining Android Sandboxes

A *sandbox* is a well-known mechanism to secure a system and forbid a software component from accessing resources without appropriate permissions. Sandboxes have also been used to build an isolated environment within which applications cannot affect other programs, the network, or other device data [17]. The idea of using sandboxes emerged from the need to test unsafe software, possible malware, without worrying about the integrity of the device under test [18], shielding the operating system from security issues. To this end, a sandbox environment should have the minimum requirements to run the program, and make sure it will never assign the program greater privileges than it should have, respecting the *least privilege* principle. Within the Android ecosystem, sandbox approaches ensure the principle of the *least privilege* by preventing apps from having direct access to resources or data from other apps. Access to sensitives resources like contacts list is granted through specific APIs (Application Programming Interface), which are managed by the Android permissions system [19].

The Mining Android Sandbox approach [5] (hereafter MAS approach) aims at automatically building a sandbox through dynamic analysis (i.e., using automatic test generation tools). The main idea is to grant permissions to an app based on its 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 the Droidmate test generation tool [20], Jamrozik et al. proposed a full fledged implementation of the MAS approach, named Boxmate [5]. Boxmate records the occurrences of calls to sensitive APIs and the UI events that triggers these calls, like button clicks. It is possible to configure Boxmate to record events associated with each sensitive call as tuples (event, API), instead of recording just the set of calls to sensitive APIs. Jamrozik et al. argue that, in this way, Boxmate generates finer grain results which might improve the accuracy of the MAS approach, even with the presence of reflection—which is quite commonly used in malicious apps [21].

In fact, the MAS approach can be implemented using a mix of static and dynamic analysis. In the first phase, one can instrument an Android app to log any call to the Android sensitive methods. After that, one can execute a test case generation tool (such as DroidBot or Monkey) to explore the app behavior at runtime, while the calls to sensitive APIs are recorded. This set of calls to sensitive APIs is then used to configure the sandbox. The general mining sandbox approach suggests that the more efficient the test generator tool (for instance, in terms of code coverage), the more accurate would be the resulting sandbox.

[2]https://anonymous.4open.science/r/paper-droidxptrace-results-F55A/

B. Mining Android Sandbox for Malware Identification

Besides being used to generate Android sandboxes, the MAS approach is also effective to detect if a repackaged version of an Android app contains malicious behavior [6]. In this scenario, the *effectiveness* of the approach is estimated in terms of the accuracy in which malicious behavior is correctly identified in the repackaged version of the apps.

The MAS approach for malware detection typically works as follows. In a first step (**instrumentation phase**), a tool instruments the code of the apps (both original and repackaged versions) to collect relevant information during the apps execution in later stages. Then, in a second step (**exploration phase**), we collect a set S_1 with all calls to sensitive APIs the original version of an app executes while running a test case generator tool (like DroidBot). In the third step (**exploitation phase**), we (a) collect a set S_2 with all calls to sensitive APIs the repackaged version of an app executes while running a test case generator tool and then (b) computes the set $S = S_2 \setminus S_1$ and check whether S is empty or not. The MAS approach classifies the repackaged version as a malware whenever $|S| > 0$.

Previous research works reported the results of empirical studies that aim to investigate the effectiveness of the MAS approach for malware identification [6], [13]. For instance, Bao et al. found that, in general, the sandboxes constructed using test generators classify at least 66% of repackaged apps as malware in a dataset comprising 102 pairs of apps (benign/repackaged) versions [6]. The authors also presented that, among five test generation tools used, DroidBot [11] leads to the most effective sandbox. Le et al. extend the MAS approach for malware detection with additional verification, such as the values of the parameters used in the calls to sensitive APIs [7], while Costa et al.[9] investigated the impact of static analysis to complement the accuracy of the MAS approach for malware detection. Their study reports that DroidFax [22], the static analysis infrastructure used in [6], is able to detect almost half of malicious code present in repackaged apps.

Our work, although closely related to previous studies, differs from them in several aspects. First, our assessment is more comprehensive: instead of considering 102 pairs of benign/malign apps, we execute our study considering 1203 pairs of apps. We then investigate which characteristics of the malware samples in the large dataset contribute to the performance of the MAS approach for malware detection. Here we also explore an extension MAS approach that compares the call graph traces from the app entry points to the calls to sensitive APIs. We discuss this extension in the next section.

C. Extending the Mining Sandbox with Trace Analysis

In order to conduct the trace analysis, we first compute a dynamic call graph (CG) that characterizes the execution of each version of the apps in our dataset. Our goal here is to explore how many pairs of apps call the same set of sensitive APIs, though using different call graph traces. We hypothesize that differences in the traces might be used to complement the MAS approach to identify suspicious behavior. As such, here we execute the trace analysis for all app pairs of our dataset, in order to reduce the number of repackaged apps incorrectly labeled as benign (false negative). For detecting different traces, we performed a comparison of the dynamic call graph of each pair of app. Our procedure checks if there is some additional CG trace from the entry points to the calls to sensitive APIs that occurs in the repackaged version but not in the original version of an app.

Figure 1 illustrates an example of benign and repackaged call graphs. In this example, although both app versions access the same set of sensitive APIs, the malicious version presents an additional execution trace.

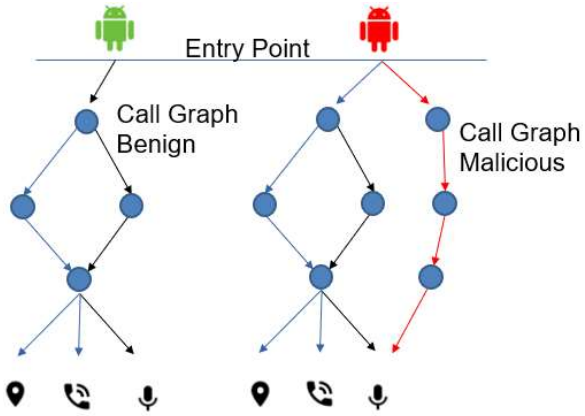

Figure 1. Illustrative example of the trace analysis. In this case, both versions call the same set of sensitive APIs. Nonetheless, the traces between the entry point and the calls to sensitive APIs diverge.

For instance, Figure 2 shows an example of a trace injected in a repackaged version of the app **[com.android.remotecontrolppt]**. Here, the original and repackaged versions access the same sensitive method, *getSubscriberid()*. This sensitive method returns the device's unique subscriber ID. The original app accesses this method through two distinct traces (Trace 01 and Trace 02), which suggests an expected use case. However, besides the two original traces, the repackaged version injects a third trace (Trace 03) containing as entry point a method that performs a stealth computation on a background thread, *doInBackground*, suggesting an action without user's awareness.

III. Experimental Setup

The goal of this research is to build an in-depth understanding about the performance of the MAS approach for detecting malware. To this end, we conduct our research using a dataset of repackaged apps one order of magnitude larger than previous studies [6], [9]. We also investigate to what extent a variant of the MAS approach with trace analysis achieves a better performance for malware detection. Altogether, our aim is to answer the following research questions:

(RQ1) What is the impact of a more diverse dataset on the accuracy of the MAS approach for malware detection?

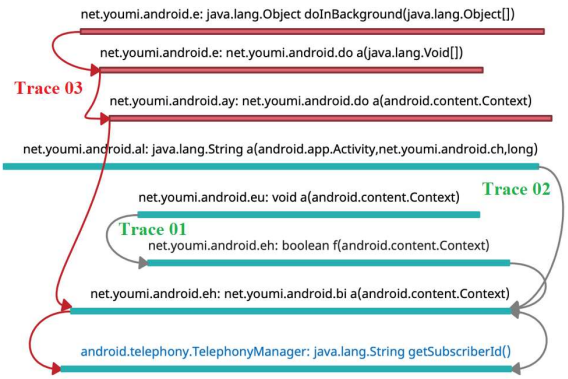

Figure 2. Example of Malicious Trace.

(RQ2) How much gain we obtain on the performance of the MAS approach for malware detection when considering trace analysis?

(RQ3) What is the influence of the similarity between the original and repackaged versions of the apps on the performance of the MAS approach for malware detection?

(RQ4) What is the influence of the malware family on the performance of the MAS approach for malware detection?

In this section, we describe our study settings. First, we present how we mined the samples of Android apps that we use as a dataset for our study (Section III-A). Then, we describe the data collection and data analysis procedures in Sections III-B and III-C.

A. Malware Dataset

We use a curated dataset of 1203 repackaged apps available in the AndroZoo repository [23]. It extends a previous dataset (hereafter `Small Dataset`) used in the research works of Bao et al. and Costa et al. [6], [13], which contains a set of 102 pairs of *original* and *repackaged* apps. We curate our dataset starting from an initial sample of 3344 repackaged pairs of apps available in AndroZoo [23]. We do not use any particular criteria for selecting the initial sample. Nonetheless, due to compatibility issues we found—either during the instrumentation phase (using DroidFax) or during the execution phase using the Android emulator—we end-up with our final dataset (hereafter `Complete Dataset`) that contains 1203 pairs of repackaged apps (36% of the initial subset repackaged apps).

We queried the `VirusTotal` repository to find out which repackaged apps in our dataset have been indeed labeled as a malware. `VirusTotal` is a well-known mechanism for scanning software assets (such as Android apps) using more than 30 anti-virus engines [24]. According to `VirusTotal`, in the `Small Dataset` (102 pairs), 69 of the repackaged apps (67.64%) have been identified as a malware by at least two security engines. Here we only consider that a repackaged version of an app is a malware if `VirusTotal` reports that at least two security engines identify a malicious behavior within the asset. This is in accordance with previous research [25], [24]. Conversely, considering the `Complete Dataset`, at least two security engine identified 459 out of the 1203 repackaged apps as malware (38.15%).

It is a common practice classify malware into different categories. For instance, Android malware can be classified into categories like riskware, trojan, adware, etc. Each category might be specialized in several malware families, depending of its characteristics and atack strategy—e.g, steal network info (IP, DNS, WiFi), collect phone info, collect user contacts, send/receive SMS, and so on [26]. According to the `avlass2` tool [27], the malware samples in the `Small Dataset` come from 17 different families—most of them from the Kuguo (49.27%) and Dowgin (17.39%) families. Contrasting, our `Complete Dataset` is more representative. Besides a large sample of repackaged apps (1203 in total), it comprises 57 families of malware we collected using the `avlass2` tool—most of them from the Gappusin (43.13%), Kuguo (9.58%), and Revmob (7.84%) families.

We also characterize our dataset according to the similarity between the original and repackaged versions of the apps, using the SimiDroid tool [28]. SimDroid quantifies the similarity based on (a) the methods that are either identical or similar in both versions of the apps (original and repackaged versions), (b) methods that only appear in the repackaged version of the apps (new methods), and (c) methods that only appear in the original version of the apps (deleted methods). Our `Complete Dataset` has an average similarity score of 62.61%—with a much better distribution (241 of app pairs have a similarity score of less than 0.25%, 156 of app pairs between 0.25% and 0.50%, 227 of the apps between 0.5% and 0.75%, and 578 of the apps with more than 0.75%). The `Small Dataset` presents a much higher similarity index average (77.87%).

It is important to highlight that our examples of apps collected at in AndroZoo come from different Android app stores. Most of our repackage apps come from a non-official Android app store, Anzhi [29]. However some repackaged apps also come from the official Android app store, Google Play.

B. Data Collection Procedures

We take advantage of the DroidXP infrastructure [13] for data collection. DroidXP allows researchers to compare test case generation tools in terms of malicious app behaviors identification, using the MAS approach. Although the comparison of test case generation tools is not the goal of this paper, DroidXP was still useful for automating the following steps of our study.

S1 **Instrumentation**: In the first step, we configure DroidXP to instrument all pairs of apps in our dataset. Here, we instrument both versions of the apps (as APK files) to collect relevant information during their execution. Under the hood, DroidXP leverages DroidFax [22] to instrument

the apps and collect static information about them. To improve the performance across multiple executions, this phase executes only once for each version of the apps in our dataset.

S2 **Execution**: In this step, DroidXP first installs the (instrumented) version of the APK files in the Android emulator we use in our experiment (API 28) and then starts a test case generation tool for executing both apps version (benign and malicious). To execute the app, our study builds upon DroidBot [11], since previous works report the best accuracy of the sandboxes built using the MAS approach and the DroidBot as test case generation tool. To also ensure that each execution gets the benefit of running on a fresh Android instance without biases that could stem out of history, DroidXP wipes out all data stored on the emulator that has been collected from previous executions.

S3 **Data Collection**: During the execution of the instrumented app, we collect all relevant information (such as calls to sensitive APIs, test coverage metrics, and so on). We use this information to analyse the performance of the MAS approach for detecting malicious behavior. In this paper we extended DroidXP with a new component (DroidXP-Trace), which allows us to investigate dynamic call graphs from the outcomes of a DroidXP execution. This was not possible using the vanilla version of DroidXP.

C. Data Analysis Procedures

We use basic statistics (average, median, standard deviation) to identify the accuracy of the MAS approach to identify malware, in both datasets we use in our research—i.e., the `Small Dataset` with 102 pairs of apps and our `Complete Dataset` with 1203 pairs. We use the Spearman Correlation [30] method and Logistic Regression [31] to understand the strengths of the associations between the similarity index of a malware with the MAS approach accuracy—that is, if the approach was able or not to identify the malicious behavior. To this particular goal, we consider both the vanilla implementation and our extended implementation of the MAS approach that executes the trace analysis. We use existing tools to reverse engineer a sample of repackaged apps in order to better understand (the lack of) accuracy of the MAS approach.

IV. Results

In this section, we detail the findings of our study. We remind the reader that our main goal with this study is to better understand the strengths and limitations of the MAS approach for malware detection using the state-of-the-art test case generation tool (DroidBot). We explore the results of our research using two datasets: the `Small Dataset` (102 pairs of apps) and the `Complete Dataset` (1203 pairs of apps).

A. Exploratory Data Analysis of Accuracy

`Small Dataset`. After running the dynamic analysis via DroidBot, our infrastructure produces a dataset with the sensitive methods that both app versions call during their execution. We consider that a test generation tool, in our case, DroidBot, builds a sandbox that labels a repackaged version of an app as a malware if there is at least one call to a sensitive method that (a) was observed while executing the repackaged version of the app and that (b) was not observed while executing the original version of the same app. If the set of sensitive methods that only the repackaged version of an app calls is empty, we conclude that the sandbox does not label the repackaged version the app as a malware. We triangulate this information with the outputs of `VirusTotal`, which might lead to one of the following situations:

- **True Positive**. The MAS approach labels a repackaged version as a malware and, according to `VirusTotal`, at least two security engines label the asset as a malware.
- **True Negative**. The MAS approach does not label a repackaged version as a malware and, according to `VirusTotal`, at most one security engine labels the asset as a malware.
- **False Positive**. The MAS approach labels a repackaged version as a malware and, according to `VirusTotal`, at most one security engine labels the asset as a malware.
- **False Negative**. The MAS approach does not label a repackaged version as a malware, and according to `VirusTotal`, at least two security engines label the asset as a malware.

Considering the `Small Dataset` (102 apps), the MAS approach for malware detection classifies a total of 69 repackaged versions as malware (67.64%). This result is close to what Bao et al. report. That is, in their original paper, the MAS approach using DroidBot classifies 66.66% of the repackaged version of the apps as malware [6]. This result confirms that we were able to reproduce the findings of the original study using our infrastructure.

Finding 1. We were able to reproduce the results of existing research using our infrastructure, achieving a malware classification in the `Small Dataset` close to what has been reported in previous studies.

In the previous studies [6], [9], the authors assume that all repackaged versions contain a malicious behavior. For this reason, the authors do not explore accuracy metrics (such as Precision, Recall, and F-measure (F_1))—all repackaged apps labeled as malware are considered true positives in the previous studies. As we mentioned, in this paper we take advantage of `VirusTotal` to label our dataset and build a ground truth: we only consider a repackaged version of an app a malware if the results of our `VirusTotal` query report that at least two security engines identify a malicious behavior in the asset. The first row of Table I shows that the vanilla MAS approach achieves an accuracy of 0.89.

We also investigate if one could improve the performance of the MAS approach using an enriched comparison approach. That is, instead of only comparing the sets of calls to sensitive APIs, here we also compare the traces from entry points to such a calls. If there is at least one trace that appears only

Table I
ACCURACY OF THE MAS APPROACH IN BOTH DATASETS.

Approach	Dataset	TP	FP	FN	Precision	Recall	F_1
Vanilla MAS approach	`Small Dataset` (102)	62	7	7	0.89	0.89	0.89
MAS approach + Traces	`Small Dataset` (102)	67	18	2	0.78	0.97	0.87
Vanilla MAS approach	`Complete Dataset` (1203)	173	173	286	0.5	0.37	0.42
MAS approach + Traces	`Complete Dataset` (1203)	214	326	245	0.39	0.46	0.42

during the execution of the test cases in the repackaged version of the app, we label that version as a malware.

As we already discussed in this section, the vanilla MAS approach fails to detect seven malware on the `Small Dataset` (FN column, first row of Table I), using DroidBot as a test case generator tool. Introducing trace analysis reduces the number of false negatives to two, with the side effect of increasing the number of false positives from 7 to 18 (see the second row of Table I). In general, the accuracy (F_1) of the MAS approach using trace analysis drops from 0.89 to 0.87.

Finding 2. Although the use of Trace Analysis reduces the number of false negatives (in comparison with the vanilla MAS approach), it slightly decreases the overall accuracy (F_1) of the MAS approach to detect malware, from 0.89% to 0.87% in the `Small Dataset`.

`Complete Dataset`. Surprisingly, when applied to our complete dataset (1203 apps), the MAS approach labels a total of 346 repackaged apps as malware (28.76% of the total number of repackaged apps)—for which the repackaged version calls at least one additional sensitive API. Our analysis also reveals a **negative result** related to the accuracy of the approach: here, the accuracy is much lower in comparison to what we reported for the `Small Dataset` (see the third row of Table I), dropping from 0.89 to 0.42 (a reduction of 47.19% in terms of F_1). This result indicates that, when considering a more representative dataset, the accuracy of the MAS approach using DroidBot drops significantly. The use of the trace analysis reduces the number of false negatives in the `Complete Dataset` (from 286 to 245), though increasing the number of false positives from 173 to 326. Altogether, the F_1 measure does not change when comparing performance of the vanilla MAS approach and its trace-based variant in the `Complete Dataset`.

Finding 3. The MAS approach for malware detection leads to a substantially lower performance on the `Complete Dataset` (1203 pairs of apps), dropping the accuracy by a factor of 47.19% in comparison to what we observed in the `Small Dataset`.

Therefore, the resulting sandbox we generate using DroidBot suffers from a significantly low accuracy rate when considering a large and more representative dataset. Besides that, enriching the MAS approach with trace analysis does not improve the overall accuracy in the `Complete Dataset` (even though we reduce the number of false negatives with trace analysis, the increasing in the number of false positives negatively impacts the performance of the approach). This is shown in the fourth row of Table I. The negative performance of the MAS approach in the `Complete Dataset` encouraged us to endorse efforts aimed at identifying potential reasons for this phenomenon and motivate the research questions RQ3 and RQ4.

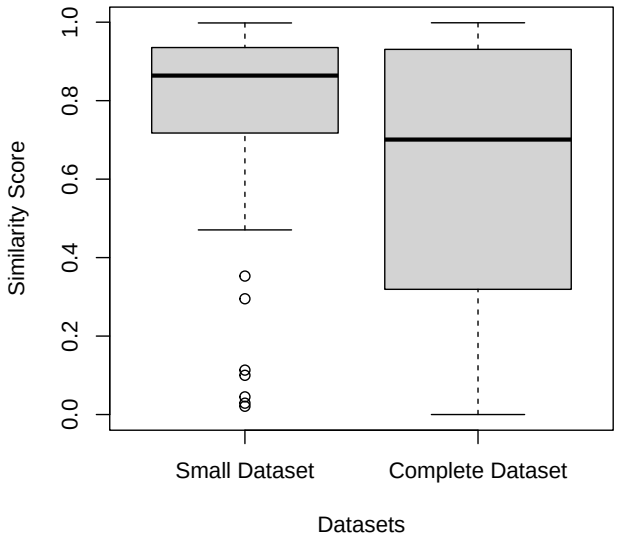

Figure 3. Similarity Score of the malware samples in the small and complete datasets.

B. Assessment Based on Similarity Score

Figure 3 shows the Similarity Score distribution over the two datasets we use in our research (the small and the complete datasets). Recall that the Similarity Score measures how similar the benign and malicious versions of an app are. The average Similarity Score of the `Small Dataset` is 0.77 (median of 0.86 and sd of 0.22). Contrasting, the average Similarity Score of the complete dataset is 0.62 (median of 0.72 and sd of 0.32).

Here we hypothesize that Similarity Score might have an association with the accuracy of the MAS approach and could perhaps explain the differences in the accuracy results we report for both datasets. We first use Logistic Regression to

assess the strength of the association between label correctness and the Similarity Score. The logistic regression results reveal a negative and real association (p-value < 0.0005). This finding suggests that the MAS approach is more likely to assign a correct label to a repackaged app in the cases where its Similarity Score with the original app is small. This is a an unexpected result, since we found a higher accuracy on the `Small Dataset` even though it presents a higher Similarity Score on average, in comparison with the `Complete Dataset`.

As a further analysis, we use the *K-Means* algorithm to split the `Complete Dataset` into five clusters, according to the Similarity Score. We then estimate the accuracy for each cluster, as we show in Table II. Note that the MAS approach achieves a percentage of hits close to 70% in the clusters 1 – 4, which present an average Similarity Score of at most 0.76.

Finding 4. We found a negative and real association between the Similarity Score and the MAS approach performance. This means that the less similar the original and repackaged versions of an app are, the more likely is the accuracy of the MAS approach to correctly label the repackaged version.

Finding 5. Besides that, the Similarity Score assessment alone does not explain the lower accuracy of the MAS approach in the `Complete Dataset`. That is, although the `Complete Dataset` presents an average Similarity Score smaller than the average Similarity Score of the `Small Dataset`, we found a lower accuracy of the MAS approach in the `Complete Dataset`.

Table II
CHARACTERISTICS OF THE CLUSTERS. NOTE THAT THE PERCENTAGE OF HITS DECREASES IN THE CLUSTER WITH THE HIGHER AVERAGE SIMILARITY SCORE.

cId	Total of Samples	Total of Hits	(%) of Hits	Similarity Score
1	196	134	68.37	0.07
2	146	101	69.18	0.29
3	182	131	71.98	0.54
4	247	175	70.85	0.76
5	431	202	46.87	0.94

C. Assessment Based on Malware Family

As we discussed in the previous section, the similarity assessment does not explain the low performance of the MAS approach on the `Complete Dataset`. Recall that the `Complete Dataset` is more heterogeneous both in terms of similarity and malware families. So, we hypothesize that the families of the malwares in the `Complete Dataset` could better explain the poor performance of the MAS approach on the `Complete Dataset`. Indeed, in the `Complete Dataset`, we identified a total of 57 families of malwares, though the most frequent ones are *gappusin* (198 samples), *kuguo* (44 samples), *revmob* (36 samples), and *dowgin* (33 samples). Together, they account for 67.75% of the repackaged apps in our complete dataset labeled as malware according to `VirusTotal`.

This family distribution in the `Complete Dataset` is significantly different from the family distribution in the `Small Dataset`—where the families *kuguo* (34 samples), *dowgin* (12 samples), and *youmi* (5 samples) account for 73.91% of the families considering the 69 repackaged apps `VirusTotal` labels as malware in the `Small Dataset`. Most important, in the `Small Dataset`, there is just one *gappusin* sample. This observation leads us to a question: *how does the MAS approach perform when considering only the gappusin samples?*

Table III shows the results of an accuracy assessment considering only those particular samples in the `Complete Dataset`. Note that for 171 samples (86.36%), neither the vanilla MAS approach nor the MAS approach with traces were able to correctly identify a repackaged version of an app as a malware. Merging the outcomes of both approaches (that is, the vanilla MAS approach and the MAS approach with traces), leads to a recall of $\frac{27}{198} = 0.13$. Further, if we remove the *gappusin* samples from the `Complete Dataset`, the recall of the MAS approach (vanilla + traces) increases to 0.71 (similar to the performance of the original studies).

Table III
ACCURACY OF THE MAS APPROACH WHEN CONSIDERING ONLY THE SAMPLES FROM THE *gappusin* FAMILY IN THE `Complete Dataset`.

Malware	Vanilla MAS approach	MAS approach + Traces	Total
True	False	False	171
True	False	True	13
True	True	False	12
True	True	True	2

Finding 6. The MAS approach fails to correctly identify 86.36% of the samples from the *gappusin* family as a malware. This is the main reason for the low recall of the MAS approach in the `Complete Dataset`.

We further analyse the sample of *gappusin* malware in our dataset, given its relevance to the negative result we present in our paper. First, Figure 4 shows a histogram of the Similarity Score for the samples in the *gappusin* family. Note that mostly of the repackaged versions from the *gappusin* family are quite similar to the original versions (average Similarity Score of 0.90, median Similarity Score of 0.95, and sd of 0.14). We also reengineer a sample of 20 *gappusin* malware (almost 10% of the samples in this family), using the `SimiDroid`[3], `apktool` [4], and `smali2java` [5] tools. Considering this sample of 20 *gappusin* malware, the median Similarity Score is 0.98. In addition, the repackaged version of the apps changes 2.73

[3]https://github.com/lilicoding/SimiDroid
[4]https://ibotpeaches.github.io/Apktool/
[5]https://github.com/AlexeySoshin/smali2java

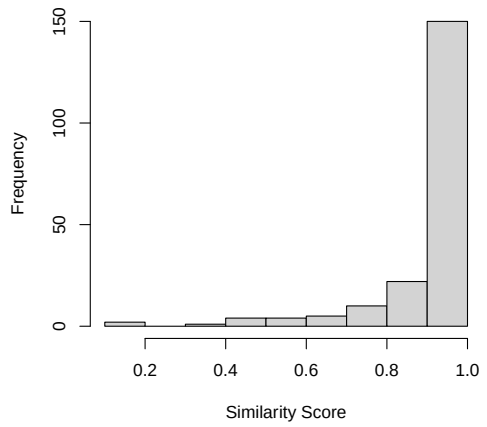

Figure 4. Histogram of the Similarity Score for the samples in the *gappusin* family.

methods, includes one new method, and delete six methods from the original app (on median). Table IV summarizes the outputs of `SimiDroid` for this sample of *gappusin* apps. The similarity assessment of this sample of 20 *gappusin* malware reveal a few modification patterns when comparing the original and the repackaged versions of the apps. First, no sample in this random selected *gappusin* dataset modifies the Android Manifest file—that is, they do not require new permissions, for instance. Moreover, 19 out of the 20 samples in this dataset **completely modifies** the method `void onReceive(Context, Intent)` of the class `com.games.AdReciver`. Although the results of the decompilation process are hard to understand (due to code obfuscation), the goal of this modification is to update the content of a `data.apk` resource. There is no extra call to sensitive API (which hinders the MAS approach to classify an asset as a malware). Figure 5 shows the code pattern of the `onReceive` method present in the samples. This particular modification typically use a new method (`public void a(Context)` the repackaged versions often introduce into the same class (`AdReciver`).

Our assessment also reveals recurrent modification patterns that **delete** methods in the repackaged version of the apps. For instance, ten repackaged apps in our *gappusin* sample of 20 malware remove methods from the class `com.game.a`. These methods extensively use the Android reflection API, and we believe that removing these methods is a strategy for antivirus evasion. For instance, although specific usages of the class `DexClassLoader` might be legitimate, it allows specific types of atack based on dynamic code injection [32]. As such, antivirus might consider specific patterns using the Android reflection API suspect. Unfortunately, the MAS approach does not identify a malicious behavior with this type of change (i.e., changes that remove methods). Listing 6 shows an example of code pattern frequently removed from the repackaged versions from the *gappusin* family.

Table IV
SUMMARY OF THE OUTPUTS OF THE `SimiDroid` TOOL FOR THE SAMPLE OF 20 *gappusin* MALWARE. (IM) IDENTICAL METHODS, (SM) SIMILAR METHODS, (NM) NEW METHODS, AND (DM) DELETED METHODS.

Hash	SimiScore	IM	SM	NM	DM
`2D76DE7`	0.99	461	1	1	6
`46C41BE`	0.99	1164	1	1	6
`0218D0E`	0.99	1082	1	1	6
`07EA86C`	0.99	2127	1	1	6
`078E0AE`	0.95	134	6	1	10
`5374927`	0.95	134	6	1	10
`17722D9`	0.97	265	6	1	10
`5B0C652`	0.99	2642	4	10	1
`CCD29EC`	0.99	436	2	0	0
`010C070`	0.99	2249	3	3	0
`723C231`	0.98	228	2	1	10
`27D5D22`	0.99	612	5	1	10
`92209D0`	0.99	698	2	3	0
`2441293`	0.98	123	2	0	11
`D83F1CE`	0.94	150	2	6	6
`00405B6`	0.99	864	2	1	10
`33896EB`	0.99	3205	2	0	0
`1AE4E8B`	0.99	3964	2	3	3
`114C5C8`	0.99	5631	2	9	151

In summary, our reverse engineering effort brings evidence that malware samples from the *gappusin* family neither modify the Android Manifest files nor call additional sensitive APIs—which reduce the ability of the MAS approach to correctly classify a sample as a malware. We argue in favor of new research efforts to integrate the MAS approach with other techniques that could increase their performance on malware identification. Since the samples from the *gappusin* family use specific patterns to introduce malicious behavior, it might be promising to explore static analysis approaches that search for these specific patterns.

V. DISCUSSION

In this section, we answer our research questions, summarize the implications of our results, and discuss possible threats to the validity of the results presented so far.

A. Answers to the Research Questions

The results we presented in the previous sections allow us to answer our four research questions, as we summarize in the following.

- **Performance of the MAS approach on the `Complete Dataset` (RQ1).** Our study indicates that the accuracy of the MAS approach reported in previous studies [6], [9] does not generalize to a larger and more diverse dataset. That is, while in our reproduction study (using the `Small Dataset` of previous research) the vanilla MAS approach leads to an accuracy of 0.89, we observed a drop of precision and recall that leads to an accuracy of 0.42 in the presence of our `Complete Dataset` (1203 pairs of original and repackaged versions of Android apps).
- **Effect of trace analysis (RQ2).** We do not find any gain of enriching the vanilla MAS approach with trace analysis

```java
public void onReceive(Context context, Intent intent) {
  SharedPreferences sharedPreferences = context.getSharedPreferences(String.valueOf("com.") + "game." + "param", 0);
  int i = sharedPreferences.getInt("sn", 0) + 1;
  System.out.println("sn: " + i);
  if (i < 2) {
    mo4a(context);
    SharedPreferences.Editor edit = sharedPreferences.edit();
    edit.putInt("sn", i);
    edit.commit();
  } else if (!new C0004b(context).f7h.equals("")) {
    String str1 = context.getApplicationInfo().dataDir;
    String str2 = String.valueOf(str1) + "/fi" + "les/d" + "ata.a" + "pk";
    String str3 = String.valueOf(str1) + "/files";
    String str4 = String.valueOf("com.") + "ccx." + "xm." + "SDKS" + "tart";
    String str5 = String.valueOf("InitS") + "tart";
    String str6 = "ff048a5de4cc5eabec4a209293513b6e";
    C0003a.m3a(context, str2, str3, str4, str5, str6);
    SharedPreferences.Editor edit2 = sharedPreferences.edit();
    edit2.putInt("sn", 0);
    edit2.commit();
  }
}
```

Figure 5. Method introduced in 19 out of 20 *gappusin* malware we randomly selected from the `Complete Dataset`.

```java
public static void m7a(Activity activity, String str, String str2, String str3, String str4, String str5) {
  try {
    Class loadClass = new DexClassLoader(str, str2, (String) null, activity.getClassLoader()).loadClass(str3);
    Object newInstance = loadClass.getConstructor(new Class[0]).newInstance(new Object[0]);
    Method method = loadClass.getMethod(str4, new Class[]{Activity.class, String.class});
    method.setAccessible(true);
    method.invoke(newInstance, new Object[]{activity, str5});
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Figure 6. Example of method that is typically removed from the repackaged apps of the *gappusin* family.

in terms of accuracy (F_1 score). Although the use of traces reduces the number of false negatives (improving recall), it also increases the number of false positives with a similar proportion—which does not change the F_1 score significantly. Nonetheless, for the context of malware identification, we believe that the gain in terms of recall might justify the use of trace analysis.

- **Similarity Analysis (RQ3).** Our results bring evidence about the existence of a negative association between the similarity of the original and repackaged versions of an app and the ability of the MAS approach to correctly classify a repackaged version of an app as a malware. Nonetheless, our similarity analysis alone was not sufficient to explain the low performance of the MAS approach to identify malware in the `Complete Dataset`.
- **Malware Family Analysis (RQ4).** The results indicate that a specific family (*gappusin*) is responsible for the largest number of false negatives in the complete dataset. The *gappusin* family corresponds to a particular type of Adware, designed to automatically display advertisements while an app is executing. After reverse engineering a sample of 20 *gappusin* malware, we confirmed that the vanilla MAS approach and its trace variant cannot identify the patterns of changes introduced in the repackaged versions of the apps. The *gappusin* family is the Achilles' heel of the MAS approach, since its accuracy increases substantially when we remove the *gappusin* samples from our `Complete Dataset`.

B. Implications

Contrasting to previous research works [6], [33], [9], our results lead to a more systematic understanding of the strengths and limitations of using the MAS approach for malware detection. In particular, this is the first study that empirically evaluates the MAS approach considering as ground truth the outcomes of `VirusTotal`. This decision allowed us to explore the MAS approach performance using well-known accuracy metrics (i.e.,m precision, recall, and F_1 score). Previous studies assume that all repackaged versions of the apps were malware. Our triangulation with `VirusTotal` reveals this is not true. Although the MAS approach presents a good accuracy

for the `Small Dataset` ($F_1 = 0.89$), in the presence of a large dataset the MAS approach accuracy drops significantly ($F_1 = 0.42$).

We also highlight that considering only the differences in the sets of calls to sensitive APIs also leads to false negatives. We argue in favor of using a more elaborate *diff* approach, which extends the mining sandbox approach to include the comparisons of the dynamic call traces from the "entry points" of an Android app to its set of calls to the sensitive APIs. Using this extension, we could reduce the number of false negatives by a factor between 14.33% (`Complete Dataset`) to 71.42% (`Small Dataset`). We also reveal that a specific family in the `Complete Dataset` is responsible for a large number of false negatives. Altogether, the takeaways of this research are twofold:

- Negative result: the mining sandbox approach for malware detection exhibits a much higher false negative rate than previous research reported.
- Future directions: researchers should advance the mining sandbox approach for malware detection by exploring more advanced techniques for differentiating benign and malicious versions of the apps. In particular, new approaches should benefit from techniques that are able to identify particular patterns of changes in the repackaged versions.

C. Threats to Validity

There are some threats to the validity of our results. Regarding **external validity**, one concern relates to the representativeness of our malware datasets and how generic our findings are. Indeed, mitigating this threat was one of the motivations for our research, since, in the existing literature, researchers had explored just one dataset of 102 pairs of benign/malign apps. Curiously, for this small dataset, the performance of the mining sandbox approach is more than twice superior than its performance on our complete dataset (1203 pairs of apps).

We contacted the authors of the Bao et al. research paper [6], asking them if they had used any additional criteria for selecting their dataset. Their answers suggest the contrary: they have not used any particular app selection process that could explain the superior performance of the mining sandbox approach for the small dataset. We believe that our results in the complete dataset generalize better than previous research works, since we have a more comprehensive collection of malwares with different families and degrees of similarity. Nonetheless, our research focus only on Android repackaged malwares and thus we cannot generalize our findings to malwares targeting other platforms or that use different approaches to generate a malicious asset.

Regarding **conclusion validity**, during the exploratory phase of the mining sandbox approach, we collected the set of calls to sensitive APIs the original version of an app executes, while running a test case generation tool (such as DroidBot). That is, the mining sandbox approach assumes the existence of a benign original version of a given app in the exploratory phase. We also query `VirusTotal` to confirm this assumption, and found that the original version of seven (out 102) apps in the `Small Dataset` contains malicious code. We believe the authors of previous studies carefully check that assumption, and this difference had occurred because the outputs of `VirusTotal` change over time [25]. Therefore, while reproducing this research, it is necessary to query `VirusTotal` to get the most up-to-date classification of the assets, which might lead to results that might slightly diverge from what we have reported here.

Regarding **construct validity**, we address the main threats to our study by using simple and well-defined metrics that are in use for this type of research: number of correctly classified samples in a dataset (true positives) and the number of assets that the classification approaches we explored here fail to correctly classify (false negatives). Based on these metrics we computed the accuracy results using precision and recall. In a preliminary study, we investigated whether or not the mining sandbox approach would classify an original version of an app as a malware, computing the results of the test case generation tools in multiple runs. After combining three executions in an original version to build a sandbox, we did not find any other execution that could wrongly label an original app as a malware.

VI. Conclusions and Future Work

The Mining Android Sandboxes (MAS) approach [20] has been tailored for Android malware detection [6] and empirically validated in a couple of studies [6], [33], [9]. To better understand the strengths and limitations of the MAS approach for malware detection, this paper reported the results of an empirical study that reproduces previous research work [6], [9] using a larger and more diverse dataset—comprising 1203 pairs of *original* and *repackaged* apps. To our surprise, compared to results already published, the performance of the MAS approach drops significantly in this new dataset, mainly because the MAS approach fails to detect a popular family of Android malware (named *gappusin*). We also evaluated an extension to the MAS approach that we designed to improve the overall accuracy of the approach for malware detection. Although this extension reduces the number of false negatives of the *vanilla MAS approach*, it was not sufficient to increase the MAS approach accuracy in the large dataset. These negative results brought evidence of the need to complement the MAS approach with other techniques, so that it could be effective for Android malware detection.

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] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, jan 2017. [Online]. Available: https://doi.org/10.1145/3017427

[5] 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

[6] 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.

[7] 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.

[8] L. Li, T. F. Bissyandé, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Trans. Software Eng.*, vol. 47, no. 4, pp. 676–693, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2901679

[9] 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

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

[11] 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

[12] N. P. B. Jr., J. Hotzkow, and A. Zeller, "Droidmate-2: a platform for android test generation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 916–919. [Online]. Available: https://doi.org/10.1145/3238147.3240479

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] H. Cai and 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. [Online]. Available: https://doi.org/10.1109/ICSME.2017.35

[23] 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

[24] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury, "Empirical study of android repackaged applications," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3587–3629, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09760-3

[25] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang, "Measuring and modeling the label dynamics of online Anti-Malware engines," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2361–2378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zhu

[26] A. Rahali, A. H. Lashkari, G. Kaur, L. Taheri, F. Gagnon, and F. Massicotte, "Didroid: Android malware classification and characterization using deep image learning," in *ICCNS 2020: The 10th International Conference on Communication and Network Security, Tokyo, Japan, November 27-29, 2020*. ACM, 2020, pp. 70–82. [Online]. Available: https://doi.org/10.1145/3442520.3442522

[27] S. Sebastián and J. Caballero, "Avclass2: Massive malware tag extraction from av labels," in *Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 42–53. [Online]. Available: https://doi.org/10.1145/3427228.3427261

[28] L. Li, T. F. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*. IEEE Computer Society, 2017, pp. 136–143. [Online]. Available: https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.230

[29] "anzhi," http://www.anzhi.com/, accessed: 2021-02-15.

[30] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904. [Online]. Available: http://www.jstor.org/stable/1412159

[31] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[32] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi, "Grab 'n run: Secure and practical dynamic code loading for android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 201–210. [Online]. Available: https://doi.org/10.1145/2818000.2818042

[33] T. B. Le, L. Bao, D. Lo, D. Gao, and L. Li, "Towards mining comprehensive android sandboxes," in *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*. IEEE Computer Society, 2018, pp. 51–60. [Online]. Available: https://doi.org/10.1109/ICECCS2018.2018.00014