The Achilles' Heel of the Android Mining Sandbox Approach for Malware Identification (Replicability Studies)

Anonymous Author(s)

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 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 1,707 pairs of apps with a much diverse malware families). 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 0.33 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.

KEYWORDS

Android Malware Detection, Dynamic Analysis, Mining Android Sandboxes

ACM Reference Format:
Anonymous Author(s). 2023. The Achilles' Heel of the Android Mining Sandbox Approach for Malware Identification (Replicability Studies). In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 and 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, comes increased risk of attacks—motivating efforts from both academia and industry to design and develop new techniques to identify malicious behavior or vulnerable code in Android apps [4].

One of the most popular classes of malwares are based on repackaging apps [5, 6] where benign versions of an app from an official app store are infected with malicious code, e.g., to broadcast sensitive information to a private server [7], and subsequently shared with users using different app stores. The focus of this paper is the Mining Android Sandbox, hereafter MAS approach, which has been shown effective in detecting the popular class of Android malware based on repackaging benign apps. It 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 a sandbox [8]. During a normal execution of the app, the sandbox might block calls to a sensitive API that had not been observed during the exploratory phase.

Previous studies [5, 9] have compared the accuracy of Android sandboxes for malware detection that were produced from different test case generation tools, including Monkey [10], DroidBot [11], and Droidmate [12] tools. The studies bring evidence that DroidBot outperforms the other test generation tools and leads to sandboxes that are more accurate in detecting malware with a classification rate of 70%. But these previous studies have two main limitations. First, they use a small dataset of malware comprising only 102 pairs of original/repackaged versions of an app. This decision might compromise the external validity of previous studies. Second, their assessment do not investigate the impact of repackaged characteristics on the accuracy of the MAS approach for malware classification, including (a) whether or not the repackaged version is a malware, (b) the similarity between the original and the repackaged versions of an app, and (c) the malware family (e.g., *gappusin*, *kuguo*, *dowgin*, etc.) when the repackaged version of an app is a malware.

To empirically study the impact of these limitations on the reported results, in this paper we reconsider the performance of the MAS approach based on DroidBot [11]—as we mentioned, the test case generation tool that, according to the literature, leads to the most accurate Android sandbox. Compared to previous studies [5, 13], we use a curated dataset of app pairs (original/repackaged versions) that is, in terms of magnitude, larger than the previously used dataset (it contains 1,707 pairs of original/repackaged apps).

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

Negative results. Our study reveals a significantly lower accuracy (F_1 score) of the MAS approach in comparison to what has been reported before (0.33 versus 0.89). An accuracy of 0.33 is clearly unsatisfactory for a trustworthy malware classification technique. This result motivated us to conduct a series of experiments to understand the reasons for the lower accuracy in our larger dataset. First, we check the impact of the similarity between original and repackaged versions of an app on the performance of the MAS approach for malware classification. Our results reveal a non significant association association between similarity and the accuracy of the MAS approach. Second, we explore if the malware family could explain the lower performance of the MAS approach for malware classification in our dataset.

Considering this second analysis, our results reveal that the MAS approach fails to correctly classify most of the samples from the *gappusin* malware family (a particular class of adware that frequently appears in repackaged apps). Out of the total of 295 samples within this family in our large dataset, the MAS approach failed to correctly classify 269 samples as malware (false negative). Our results reveal that this particular family is responsible for substantially reducing the recall of the MAS approach. Our findings have two main implications, which open the discussion (a) on one important feature found at *gappusin* malware family that should be considered when building malware detection approaches that mine sandboxes, and (b) on the need to have a representative dataset, that should be labeled with sought answers and which should leads close to real-life scenarios.

The remainder of this paper is organized as follows. We provide background information on the MAS approach and the MAS approach for malware detection in Section 2. Section 3 characterizes our experiments in terms of research goal, questions, metrics, datasets, and our procedures for data collection and data analysis. Section 4 and Section 5 present the main findings of our experiments and possible threats to the validity of our results. Finally, Section 6 presents concluding remarks and possible future work. The main artifacts we produced during this research are available in the paper repository.

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

2 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, including the MAS approach for malware classification [5, 8].

2.1 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 like device hardware (GPS, Camera), or sensitive data from other apps. Access to sensitives data like contacts list or resources are granted through specific APIs (Application Programming Interface), known as sensitive APIs, which are managed by by coarse-grained Android permissions system [19].

The MAS approach [8] 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 [8]. Boxmate records the occurrences of calls to sensitive APIs from native code, 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 MAS 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.

2.2 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 [5]. 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 classification 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 classification [5, 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 (original/repackaged versions) [5]. Actually, the mentioned work performed two studies: one pilot study involving a dataset of 10 pairs of apps (`SmallE`), in which the authors executed each test case generation tools for one hour; and a larger experiment (`LargeE`) involving 102 pairs of apps in which the authors executed the test case generation tools for one minute [5].

Here we replicate their larger experiment. 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 classification with additional verification, such as the values of the parameters used in the calls to sensitive APIs [6], while Costa et al.[9] investigated the impact of static analysis to complement the accuracy of the MAS approach for malware classification. Their study reports that DroidFax [22], the static analysis infrastructure used in [5], classifies as malware almost half of the 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 original/repackaged apps, we execute our study considering 1,707 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 classification.

3 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 [5, 9]. Altogether, explore the the following research questions:

(RQ1) What is the impact of considering a larger diverse dataset on the accuracy of the MAS approach for malware classification?
(RQ2) 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 classification?
(RQ3) What is the influence of the malware family (e.g., *gappusin*, *kuguo*, *dowgin*) on the performance of the MAS approach for malware classification?

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 3.1). Then, we describe the data collection and data analysis procedures (Sections 3.2 and 3.3). Finally, we detail our experiment configuration on Section 3.4.

3.1 Malware Dataset

To investigate our research questions, we shall run our infrastructure on a representative dataset and estimate the MAS approach performance (in terms of accuracy). Our dataset shall also be *labeled*, i.e., the interest characteristic of each app, like similarity and malware family, should be known beforehand.

3.1.1 Procedures for Building the Dataset. Figure 1 presents the methodology we use to extract the dataset for our study. We start with an original dataset containing 15,297 pairs of original/repackaged Android apps [7]. This original dataset has been curated using automatic procedures that mine repackaged apps from the well-known Androzoo repository [23]. From this original dataset, we filter a random sample of 5,700 pairs. However, among these 5,700 app pairs, we fail to execute our study in 3,381 samples. We could not either instrument some of these samples using DroidFax [22] or we fail to install some of them on the emulator—due to compatibility issues with the Android SDK. After removing incompatible apps, we were left with 2,319 samples. To build our final dataset (hereafter `Complete Dataset`), we queried the `VirusTotal` repository to find out which original version of the apps have been labeled as a malware. We exclude these samples from our dataset—since the MAS approach assumes that the original version of an app is legitimate. `VirusTotal` is a well-known mechanism for scanning software assets (such as Android apps) using more than 60 anti-virus engines [24]. In the end, we are left with our `Complete Dataset` of 1,707 apps which we use for this study. In our research, we also consider a small dataset (hereafter `Small Dataset`) used in previous studies [5, 9].

3.1.2 Features of the Datasets. We queried the `VirusTotal` repository to find out which repackaged apps in our dataset have been indeed labeled as a malware. 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 [24, 25]. Conversely, considering the `Complete Dataset`, at least two security engine identified 490 out of the 1,707 repackaged apps as malware (28.70%).

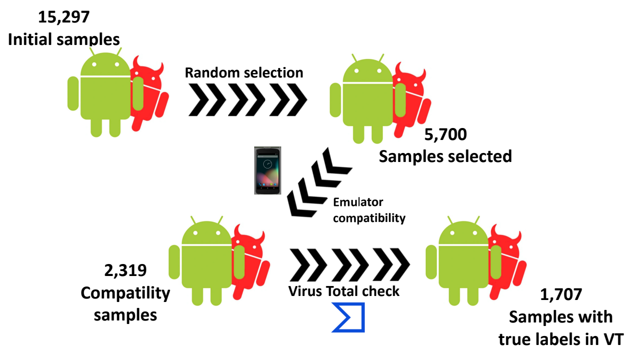

Figure 1: Malware samples in the Complete Datasets.

Classifying malware into different categories is a common practice. For instance, Android malware can be classified into categories like riskware, trojan, adware, etc. Each category might be further 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 `avclass2` 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. Our `Complete Dataset`, besides a large sample of repackaged apps (1,707 in total), comprises 44 families of malware we collected using the `avclass2` tool—most of them from the Gappusin (60.2%) family.

We also characterize our dataset according to the similarity between the original and repackaged versions of the apps, using the SimiDroid tool [28]. SimiDroid 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 93.37%—with the follow distribution (27 of app pairs have a similarity score of less than 25%, 18 of app pairs between 25% and 50%, 55 of the apps between 50% and 75%, and 1607 of the apps with more than 75%). The `Small Dataset` presents a lowest similarity index average (89.41%).

After executing our experiments, we identified the most frequently abused sensitive APIs called by the repackaged version of our samples. We observed that upon execution of all samples from the `Complete Dataset`, malicious app versions injected 138 methods from sensitive APIs (according to the AppGuard [29] security framework). Malicious code may use these APIs to compromise system security and share sensitive data. Table 1 presents a list of methods from sensitive APIs the malicious versions of the apps frequently inject.

It is important to highlight that our sample comes from different Android app stores. Most of our repackage apps come from a non-official Android app store, Anzhi [30]. However some repackaged apps also come from the official Android app store, Google Play.

3.2 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 app versions (original and repackage). We execute the apps via DroidBot [11], mostly because previous research 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 apps, 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.

3.3 Data Analysis Procedures

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 APIs 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 of an app as a malware. The set of sensitive APIs used at our work was defined in the AppGuard framework [29], which was based on the mapping from sensitive APIs to permissions proposed by Song et al. [31]. We triangulate this information with the outputs of `VirusTotal`, which might lead to one of the following situations:

- **True Positive (TP)**. 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 (TN)**. 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 (FP)**. 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.

Table 1: Sensitive APIs that frequently appear in the repackaged versions of the apps. The *Occurrences* column gives the number of distinct repackaged apps that introduce a call to a sensitive method.

Method of Sensitive API	Occurrences
android.net.wifi.WifiInfo: java.lang.String getMacAddress()	151
android.net.NetworkInfo: android.net.NetworkInfoState getState()	150
android.net.wifi.WifiManager: android.net.wifi.WifiInfo getConnectionInfo()	149
android.telephony.TelephonyManager: java.lang.String getSubscriberId()	147
android.telephony.TelephonyManager: java.lang.String getDeviceId()	142
android.net.ConnectivityManager: android.net.NetworkInfo getNetworkInfo(int)	130
android.telephony.TelephonyManager: java.lang.String getNetworkOperator()	127
android.net.NetworkInfo: java.lang.String getTypeName()	125

- **False Negative (FN)**. 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.

We compute *Precision*, *Recall*, and *F-measure* (F_1) from the number of true-positives, false-positives, and false-negatives (using standard formulae). We use basic statistics (average, median, standard deviation) to identify the accuracy of the MAS approach for malware classification, in both datasets we use in our research—i.e., the `Small Dataset` with 102 pairs of apps and our `Complete Dataset` with 1,707 pairs. We use the Spearman Correlation [32] method and Logistic Regression [33] to understand the strengths of the associations between the similarity index between the original and the repackaged versions of a malware with the MAS approach accuracy—that is, if the approach was able to correctly classify an asset as malware. We also use existing tools to reverse engineer a sample of repackaged apps in order to better understand (the lack of) accuracy of the MAS approach.

3.4 Environment Configuration

We deployed our experiment on a 32-Core, AMD EPYC 7542 CPU, 512 GB RAM, storage Samsung SSD 970 EVO 1TB machine running a 64-bit Debian GNU/Linux 11. We also configured our emulator to run all selected apps on Google Android version 9.0, API 28, 512M SD Card, 7GB internal storage, with X86 ABI image. For our study, we configured DroidXP to run each of the 1,707 app pairs using DroidBot for 3 minutes. To mitigate noise, we repeated the full process 3 times, which took around 513 machine hours in total. Although it was possible to run more than 10 emulators in parallel on one physical machine, to avoid any interference resulting from context switching within the operating system, we chose to run one emulator at a time. Hence, all processes took around 27 days, 23 days for experiment execution and additional 4 days for environment configuration.

4 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` (1,707 pairs of apps).

4.1 Exploratory Data Analysis of Accuracy

`Small Dataset`. 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 [5]. That is, in their original paper, the MAS approach using DroidBot classifies 66.66% of the repackaged version of the apps as malware [5]. 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 [5, 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 classify a repackaged version of an app as malware if the results of our `VirusTotal` query report that at least two security engines identify malicious behavior in the asset (again, this decision follows existing recommendations [24, 25]). The first row of Table 2 shows that the MAS approach achieves an accuracy of 0.89 when considering the `Small Dataset`. Nonetheless, the MAS approach fails to correctly classify seven assets as malware on the `Small Dataset` (FN column, first row of Table 2), and wrongly labeled the repackaged version of seven apps as malware (FP column).

`Complete Dataset`. Surprisingly, considering our complete dataset (1,707 apps), the MAS approach labels a total of 398 repackaged apps as malware (23.31% 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 second row of Table 2): F_1 dropping from 0.89 to 0.33. This result indicates that, when considering a large dataset, the accuracy of the MAS approach using DroidBot drops significantly.

Table 2: Accuracy of the MAS approach in both datasets.

Dataset	TP	FP	FN	Precision	Recall	F_1
`Small Dataset` (102)	62	7	7	0.89	0.89	0.89
`Complete Dataset` (1,707)	149	249	341	0.37	0.30	0.33

Finding 2. The MAS approach for malware detection leads to a substantially lower performance on the `Complete Dataset` (1,707 pairs of apps), dropping the accuracy by a factor of 56% 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 dataset. This is shown in the second row of Table 2. 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 RQ2 and RQ3.

4.2 Assessment Based on Similarity Score

Figure 2 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 original and repackaged versions of an app are. The average Similarity Score of the `Small Dataset` is 0.89 (median of 0.99 and sd of 0.25). Contrasting, the average Similarity Score of the complete dataset is 0.93 (median of 0.98 and sd of 0.14).

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. To this analysis, we removed the occurrences of true negatives (i.e., the repackaged version is benign according to `VirusTotal` and the MAS approach correctly label the repackaged version as benign). As such, we test the following null hypothesis:

H_0 Similarity Score does not influence the accuracy of the MAS approach for malware detection.

The results of the logistic regression suggest that we should not reject our null hypothesis (p-value = 0.89). This finding suggests that the MAS approach is not more likely to assign a correct label to a repackaged app in the cases where its Similarity Score with the original app is high (or small). This is an expected result, since we found a higher accuracy on the `Small Dataset`, even though the Similarity Score of the original and repackage apps in both datasets do not differ significantly.

Finding 3. There is no significant association between the Similarity Score and the MAS approach performance, which means that the similarity between the original and repackaged versions of an app is not sufficient to explain the performance of the MAS approach for malware classification.

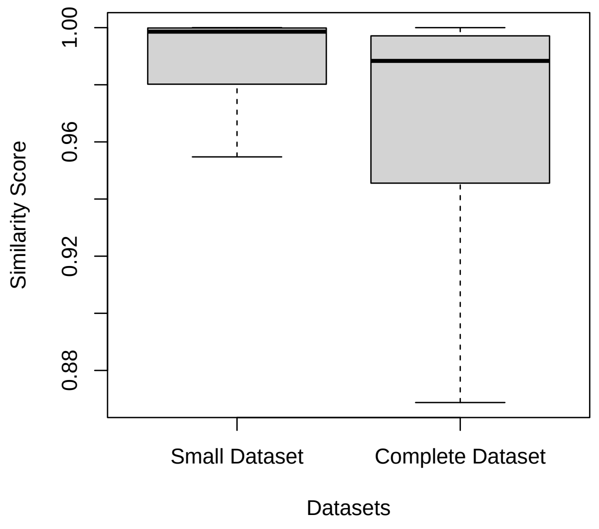

Figure 2: Similarity Score of the malware samples in the small and complete datasets. The boxplots in the figure do not show outliers.

To clarify the lack of association between Similarity Score and correctness, we use the *K-Means* algorithm to split the `Complete Dataset` into ten clusters—according to the Similarity Score. We then estimate the percentage of correct classifications for each cluster, as we show in Table 3. Note that the MAS approach achieves the highest percentage of correct classification (40.54%) for the sixth cluster (cId = 6), which presents an average Similarity Score of 0.91%. Nonetheless, the cluster cId = 10, with larger number of samples (316) and Similarity Score (99%), presents a percentage of correct classifications around 22%.

4.3 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`. Since the `Complete Dataset` covers a wide range of malware families, we investigate the hypothesis that the malware families in the `Complete Dataset` could explain the poor performance of the MAS approach on the `Complete Dataset`. Indeed, in the `Complete Dataset`, we identified a total of 44 families

Table 3: Characteristics of the clusters. Note there is no specific pattern associating the percentage of correct answers with the Similarity Score. For this analysis, we removed the true negatives in our dataset.

cId	Similarity Score	Samples	Correct Answers	Percentage
1	0.02	10	1	10.00
2	0.49	13	2	15.38
3	0.71	14	5	35.71
4	0.80	45	11	24.44
5	0.88	68	10	14.71
6	0.91	37	15	40.54
7	0.94	48	8	16.67
8	0.97	44	8	18.18
9	0.98	144	19	13.19
10	0.99	316	70	22.15

of malwares, though the most frequent ones are *gappusin* (295 samples), *revmob* (31 samples), *dowgin* (25 samples), and *airpush* (21 samples). Together, they account for 75.91% of the repackaged apps in our complete dataset labeled as malware according to `VirusTotal`.

This family distribution in the `Complete Dataset` is 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 sample from the *gappusin* family. This observation leads us to the question: *how does the MAS approach perform when considering only the gappusin samples?*

The confusion matrix of Table 4 summarizes the accuracy assessment of the MAS approach considering only the *gappusin* samples in the `Complete Dataset`. Note that `VirusTotal` classifies as malware all repackaged versions in the *gappusin* family and that for 269 samples (91.18%), the MAS approach was not able to correctly identify a repackaged version of an app as a malware (recall of 0.08). Further, if we remove the *gappusin* samples from the `Complete Dataset`, the recall of the MAS approach increases to 0.63 (similar to the performance of the original studies).

Table 4: Confusion matrix of the MAS approach when considering only the samples from the *gappusin* family in the `Complete Dataset`.

Actual Condition	Predicted Condition	
	Benign	Malware
Benign (0)	TN (0)	FP (0)
Malware (295)	FN (269)	TP (26)

Finding 4. The MAS approach fails to correctly identify 91.18% 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 3 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 0.97, median Similarity Score of 0.99, and sd of 0.06). We also reengineer a sample of 30 *gappusin* malware (more than 10% of the samples in this family), using the `SimiDroid`[2], `apktool`[3], and `smali2java`[4] tools. Considering this sample of 30 *gappusin* malware, the median Similarity Score is 0.99. In addition, the repackaged version of the apps changes 4.96 methods (on average). Table 5 summarizes the outputs of `SimiDroid` for this sample of *gappusin* apps.

The similarity assessment of this sample of 30 *gappusin* malware reveals a few modification patterns when comparing the original and the repackaged versions of the apps. First, no instance in this *gappusin* sample dataset, modifies the Android Manifest file to require additional permissions, for instance. In most of the cases, the repackaged version just changes the Manifest file to modify either the package name or the main activity name. Moreover, 29 out of the 30 samples in this dataset **modifies** the method `void onReceive(Context,Intent)` of the class `com.games.AdReciver`. Although the results of the decompilation process are difficult to understand in full (due to code obfuscation), the goal of this modification is to change the behaviour of the benign version, so that it can download a different version of the `data.apk` asset. Figure 4 shows the code pattern of the `onReceive` method presents in the samples. This modification typically use a new method (`public void a(Context)` the repackaged versions often introduce into the same class (`AdReciver`). Since there is no extra call to sensitive APIs, the MAS approach fails to label the *gappusin* samples correctly.

Our assessment also reveals recurrent modification patterns that **delete** methods in the repackaged version of the apps. For instance, 20 repackaged apps in our *gappusin* sample of 30 malware remove the method `void b(Context)` from the class `com.game.a`. This class extensively use the Android reflection API. Although it is not clear the real purpose of removing these methods, that decision simplifies the procedure of downloading a `data.apk` asset that is different from the asset available in the original version of the apps. Removing those methods might also be an strategy for antivirus evasion. For instance, although some usages of the class `DexClassLoader` might be legitimate, it allows specific types of atack based on dynamic code injection [34]. As such, antivirus might flag specific patterns using the Android reflection API suspect. Unfortunately, the MAS approach also fails to identify a malicious behavior with this type of change (i.e., changes that remove methods). Listing 5 shows an example of code pattern frequently removed from the repackaged versions from the *gappusin* family.

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. Both versions (original/repackaged) from the *gappusin*

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

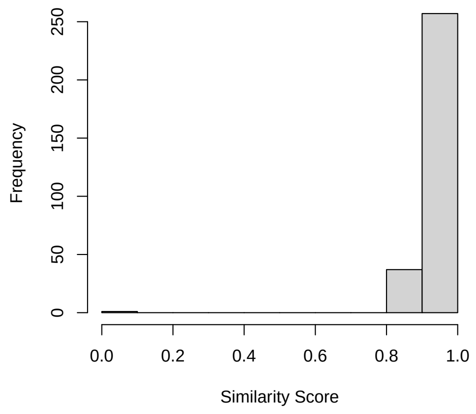

Figure 3: Histogram of the Similarity Score for the samples in the *gappusin* family.

family have the same behavior for showing advertisements to the user, however the repackaged version have additional call sites to the advertisement API and the advertisements sources are different.

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.

5 DISCUSSION

In this section, we answer our research questions, summarize the implications of our results, and discuss possible limitations of our study that might threaten the validity of the results presented so far.

5.1 Answers to the Research Questions

The results we presented in the previous sections allow us to answer our three 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 [5, 9] does not generalize to a larger dataset. That is, while in our reproduction study (using the `Small Dataset` of previous research) the MAS approach leads to an accuracy of 0.89, we observed a drop of precision and recall that leads to an accuracy of 0.33 in the presence of our `Complete Dataset` (1,707 pairs of original and repackaged versions of Android apps).
- **Similarity Analysis (RQ2).** Our results bring evidence about the nonexistence of an 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. Therefore, the similarity assessment does not explain the low performance of the MAS approach to identify malware in the `Complete Dataset`.
- **Malware Family Analysis (RQ3).** 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 running. After reverse engineering a sample of 30 *gappusin* malware, we confirmed that the MAS approach cannot identify the patterns of changes introduced in the repackaged versions of the apps. The prevalence of the *gappusin* family in the `Complete Dataset` largely explains the poor performance of the MAS approach for malware classification in the large dataset.

Table 5: Summary of the outputs of the `SimiDroid` tool for the sample of 30 *gappusin* malware. (IM) Identical Methods, (SM) Similar Methods, (NM) New Methods, and (DM) Deleted Methods.

Hash	Similarity Score	IM	SM	NM	DM
33896E	0.9994	3205	2	0	0
0C962D	0.9994	3413	1	1	10
BCDF91	0.9992	2645	2	0	0
01ECE4	0.9991	5697	4	1	10
A306DA	0.9989	1886	1	1	6
4010CA	0.9987	3721	1	4	6
5B5F2D	0.9983	1164	2	3	0
010C07	0.9982	2248	4	3	0
F9FC04	0.9982	1121	1	1	6
E29F53	0.9976	842	1	1	6
FE76EB	0.9976	839	1	1	6
842BD5	0.9973	2249	3	3	3
295B66	0.9972	1081	2	1	10
92209D	0.9971	698	2	3	0
0977B0	0.9969	1613	4	1	10
347FCF	0.9967	613	1	1	6
00405B	0.9965	864	2	1	10
67310E	0.9957	1164	2	3	3
CCD29E	0.9954	436	2	0	0
610113	0.9941	836	4	1	10
A871E0	0.9941	836	4	1	10
ECEA10	0.9913	229	1	1	6
E53FAA	0.9889	267	2	1	10
723C23	0.9870	228	2	1	10
D95B6E	0.9870	833	10	1	10
17722D	0.9743	265	6	1	10
537492	0.9504	134	6	1	10
078E0A	0.9504	134	6	1	10
D83F1C	0.9494	150	2	6	6
E5D716	0.8840	2035	68	199	199

Although the *gappusin* family explains the drop in the recall of the MAS approach, other reasons might explain the reduced precision of the MAS approach in the `Complete Dataset`. First, the

```
public void onReceive(Context context, Intent intent) {
  SharedPreferences sp = context.getSharedPreferences(String.valueOf("com.") + "game." + "param", 0);
  int i = sp.getInt("sn", 0) + 1;
  System.out.println("sn: " + i);
  if (i < 2) {
    mo4a(context);
    SharedPreferences.Editor edit = sp.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 = sp.edit();
    edit2.putInt("sn", 0);
    edit2.commit();
  }
}
```

Figure 4: Method introduced in 29 out of 30 *gappusin* malware we randomly selected from the `Complete Dataset`.

```
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 5: Example of method that is typically removed from the repackaged apps of the *gappusin* family.

proportion of malware samples in the datasets significantly differs. That is, `VirusTotal` labels only 40.26% of the repackaged version of the apps in the `Complete Dataset` as malware—contrasting with 67.64% of the samples that `VirusTotal` labels as malware in the `Small Dataset`. The lack of balance in the datasets might partially explain the low precision of the MAS approach in the `Complete Dataset`. Second, we only label a repackaged version of an app as malware if `VirusTotal` reports that at least two engines find suspicious behavior in that asset. This decision might be considered an over-constraint. When we relax this constraint and label an assets as malware whenever at least one engine finds suspicious behavior, the precision of the MAS approach increases from 0.37 to 0.59—still significantly smaller than the MAS approach precision for the `Small Dataset`.

5.2 Implications

Contrasting to previous research works [5, 9, 35], our results lead to a more systematic understanding of the strengths and limitations of using the MAS approach for malware classification. 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., 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 = 0.33).

We also reveal that a specific family in the `Complete Dataset` is responsible for a large number of false negatives, compromising the accuracy of the MAS approach. Altogether, the takeaways of this research are twofold:

- Negative result: the MAS approach for malware detection exhibits a much higher false negative rate than previous research reported.
- Future directions: researchers should advance the MAS 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.

5.3 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 original/repackaged apps. Curiously, for this small dataset, the performance of the MAS approach is more than twice superior than its performance on our `Complete Dataset` (1,707 pairs of apps).

We contacted the authors of the Bao et al. research paper [5], asking them if they had used any additional criteria for selecting the pairs of apps in their dataset. Their answers suggest the contrary: they have not used any particular app selection process that could explain the superior performance of the MAS 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 malware with different families and degrees of similarity. Nonetheless, our research focus only on Android repackaged malware and thus we cannot generalize our findings to malware that targets other platforms or that use different approaches to generate a malicious asset.

Regarding **conclusion validity**, during the exploratory phase of the MAS approach, we collected the set of calls to sensitive APIs the original version of an app executes, while running a test case generation tool (DroidBot). That is, the MAS 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], and a dataset that is consistent at a date will not so in the future. 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. Besides that, in the `Complete Dataset` we only consider pairs of original/repackaged apps for which `VirusTotal` classifies the original version as benign.

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 malware samples the MAS approach correctly/wrongly classify in a dataset (true positives/-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 MAS 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.

6 CONCLUSIONS

To better understand the strengths and limitations of the MAS approach for repackaged malware detection, this paper reported the results of an empirical study that reproduces previous research works [5, 9] using a larger and more diverse dataset—in comparison to the datasets used in previous research. To our surprise, compared to published results, the performance of the MAS approach drops significantly for our comprehensive dataset. This result is largely explained by the prevalence of a specific malware family (named *gappusin*), for which the MAS approach fails to correctly classify. We also report the results of a reverse engineering effort, whose goal was to understand the features from the *gappusin* family that reduce the performance of the MAS approach for malware classification. Our reverse engineering effort revealed common changing patterns in the *gappusin* repackaged versions of original apps, which mostly use reflection to download an external apk asset for handling advertisements without introducing additional calls to sensitive APIs. This kind of change compromises the ability of the MAS approach for malware identification. These negative results showed how far we are for using the MAS approach for malware classification.

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] 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.

[6] 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.

[7] 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

[8] 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

[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] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard - fine-grained policy enforcement for untrusted android applications," in *Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM 2013, and 6th International Workshop, SETOP 2013, Egham, UK, September 12-13, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. García-Alfaro, G. V. Lioudakis, N. Cuppens-Boulahia, S. N. Foley, and W. M. Fitzgerald, Eds., vol. 8247. Springer, 2013, pp. 213–231. [Online]. Available: https://doi.org/10.1007/978-3-642-54568-9_14

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

[31] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 627–638. [Online]. Available: https://doi.org/10.1145/2046707.2046779

[32] 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

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

[34] 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

[35] 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