Scaling Up: Revisiting Mining Android Sandboxes at Scale for Malware Classification

Francisco Handrick, Ismael Medeiros, Leandro Oliveira, João Calássio, Rodrigo Bonifácio, Krishna Narasimhan, Mira Mezini, Márcio Ribeiro

✦

Abstract—The widespread use of smartphones in daily life has raised concerns about privacy and security among researchers and practitioners. Privacy issues are generally highly prevalent in mobile applications, particularly targeting the Android platform—the most popular mobile operating system. For this reason, several techniques have been proposed to identify malicious behavior in Android applications, including the Mining Android Sandbox approach (MAS approach), which aims to identify malicious behavior in repackaged Android applications (apps). However, previous empirical studies evaluated the MAS approach using a small dataset consisting of only 102 pairs of original and repackaged apps. This limitation raises questions about the external validity of their findings and whether the MAS approach can be generalized to larger datasets. To address these concerns, this paper presents the results of a replication study focused on evaluating the performance of the MAS approach regarding its capabilities of correctly classifying malware from different families. Unlike previous studies, our research employs a dataset that is an order of magnitude larger, comprising 4,076 pairs of apps covering a more diverse range of Android malware families. Surprisingly, our findings indicate a poor performance of the MAS approach for identifying malware, with the F1-score decreasing from 0.89 for the small dataset used in the previous studies to 0.54 in our more extensive dataset. Upon closer examination, we discovered that certain malware families partially account for the low accuracy of the MAS approach, which fails to classify a repackaged version of an app as malware correctly. Our findings highlight the limitations of the MAS approach, particularly when scaled, and underscore the importance of complementing it with other techniques to detect a broader range of malware effectively. This opens avenues for further discussion on addressing the blind spots that affect the accuracy of the MAS approach.

1 INTRODUCTION

Mobile technologies, such as smartphones and tablets, have become fundamental to how 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 2.5 million Android applications [1] (apps) available on the Google Play Store, in June 2023 [3]. As popularity rises, so does the risk of potential attacks, prompting collaborative efforts from both academia and industry to design and develop new techniques for identifying malicious behavior or vulnerable code in Android apps [4]. One popular class of Android malware is based on repackaging [5], [6], where a benign version of an app is infected with malicious code, e.g., to broadcast sensitive information to a private server [7], and subsequently shared with users using even official app stores.

The Mining Android Sandbox approach (MAS approach) was initially designed to construct sandboxes based on exploratory calls to sensitive APIs [8]. The MAS approach operates in two distinct phases. In the first phase (exploratory phase), automated test case generation tools are utilized to abstract the behavior of an app, focusing on recording calls to sensitive APIs (Application Programming Interfaces). Subsequently, during normal app execution (exploitation phase), the generated sandbox blocks any calls to sensitive APIs that were not observed during the exploratory phase. Prior studies [5], [9] have investigated the effectiveness of the MAS approach in detecting potential malicious behavior in repackaged apps. These studies have also conducted comparisons of the approach's performance by employing different test case generation tools during the exploratory phase, including Monkey [10], DroidBot [11], and Droidmate [12]—bringing evidence that DroidBot outperforms other test generation tools, uncovering many potential malicious behaviors.

Nonetheless, 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—which might compromise external validity. Second, their assessments do not investigate the impact of relevant features of the repackaged apps 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 [2] when the repackaged version of an app is a malware. These limitations compromise a broader understanding of the MAS approach performance. We present more details about the MAS approach and related work in Section 2.

To better understand the impact of these issues on pre-

---

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

2. Malware families (such as *gappusin*, *kuguo*, *dowgin*, etc.) are often used to classify malware in groups that share similar codebases, attack methods, and objectives [13].

viously published results, this paper presents a replication of the study conducted by Bao et al. [5]. We aim to verify the original study's findings by executing the test case generation tool DroidBot [11] in the same settings as the original research. Unlike the original study, here we use a curated dataset of app pairs (original/repackaged versions) significantly larger than the dataset used in Bao et al.'s study. Our new dataset contains 4,076 pairs of original and repackaged apps. We present more details about the datasets, data collection, and analysis procedures in Section 3.

Negative result. Our study reveals a significantly lower accuracy (F1-score of 0.54) of the MAS approach in comparison to what the MAS approach approach performs in the small dataset (F1-score of 0.89). Since an accuracy of 0.54 is unsatisfactory for a trustworthy malware classification technique, we conduct a set of experiments to understand the reasons for the lower accuracy in our dataset. Our further assessments reveal that the MAS approach fails to correctly classify most samples from a specific set of malware families, particularly those from the *gappusin* family (a particular adware class frequently appearing in repackaged apps). Out of the total of 1,337 samples within this family in our large dataset, the MAS approach failed to classify 1,170 samples as malware correctly. Accordingly, these families are responsible for substantially reducing the recall of the MAS approach. We detail the results of our experiments in Section 4. We also discuss the implications and possible threats to the validity of our study in Section 5 and present some final remarks in Section 6. The main artifacts we produced during this research are available in the paper repository.

https://github.com/droidxp/paper-ecoop-results

2 BACKGROUND AND RELATED WORK

There are many tools available that help developers reverse engineer Android bytecode [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 their propagation and spread malware [7]. As an example, in 2016 a repackaged version of the famous Pokémon Go app was discovered less than 72 hours after the game was officially released in the United States, Australia, and New Zealand [7]. The repackaged version, originated from an unofficial app store, gained full control over the victim's phone, obtaining access to main functions such as the phonebook, audio recorder, and camera.

Repackaging has been raised as a noteworthy security concern in Android ecosystem by stakeholders in the app development industry and researchers [15]. Indeed, there are reports claiming that about 25% of Google Play Store app content correspond to repackaged apps [16]. 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 [17]. 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 [8], [5].

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 [18]. 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 [19], 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 more 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 (e.g., GPS, Camera), or sensitive data from other apps. Access to sensitives data like contact list or resources are granted through specific APIs, known as sensitive APIs, which are managed by coarse-grained Android permissions system [20].

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 [21], Jamrozik et al. proposed a full-fledged implementation of the MAS approach, named Boxmate [8]. Boxmate records the occurrences of calls to sensitive APIs and, optionally, the UI events (e.g., a button click) that trigger these calls. Therefore, 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, a feature commonly used in malicious apps [22].

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 DroidMate, 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 generation tool (for instance, code coverage), the more accurate the resulting sandbox would be.

2.2 Mining Android Sandbox for Malware Classification

Besides being used to generate Android sandboxes, the MAS approach can also be used to detect if a repackaged version of an Android app contains an unexpected (perhaps 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**), the MAS approach collects 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**), the MAS approach (a) collects 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 checks 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], [23]. For instance, Bao et al. found that, in general, the sandboxes constructed using test generation tools 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].

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 actual 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 [24], the static analysis infrastructure used in [5], classifies as malware almost half of the repackaged apps.

2.3 Android Malware Classification

The field of malware detection for the Android platform is fertile, with a significant number of secondary studies already published [25], [26], [27], [28]. In general, malware detection techniques are divided into static detection, dynamic detection, and hybrid detection [29]. Several studies have also conducted surveys on malware detection techniques and presented a review of them [30], [31], [32]. For instance, M. Odusami et al. [32] discuss various static analyses approaches that have been used in the literature to identify malicious behavior in Android apps. The authors present some works with permission and signature-based malware detection systems. They highlight that both approaches have a low false positive rate; however, they are very ineffective in detecting new malware. Although they could reveal possible malicious behaviors, the authors discuss several limitations of these approaches, as they are limited regarding code obfuscation and dynamic code loading.

The literature also presents surveys based on dynamic analysis, where malicious behavior is analyzed at runtime, exposing risks that are not detected by static analysis. As a malicious app "is alive", dynamic analysis adds another degree of analysis since it observes how Android apps interacts with the environment. However, if applied inappropriately, it may provide limited code coverage, which repeated executions can improve. Therefore, dynamic analysis's time cost and computation resources are higher when compared with static analysis. K. Tam et al. [31] presented several dynamic analysis studies based on Android architectural layers. The survey also exposed that dynamic analysis can be performed in emulator environments, real devices, or both. The authors discuss that the choice of environments is an important issue for analysis, as there are malware families that can detect emulated environments and do not exhibit malicious behaviors [33]. Finally, K. Tam et al. also exposed some works based on hybrid malware detectors and claim that these methods can increase code coverage and robustness, taking advantage of the best of each technique to find malicious behaviors.

Several studies have also explored Android malware detection approaches based on machine learning (ML) techniques, employing both static and dynamic analyses to extract features and train ML models [30]. Most of these approaches have demonstrated high accuracy (above 90%), effectively detecting previously unseen malware families with low false positives rate [34]. However, some studies have identified limitations of machine learning approaches for Android malware classification. In their work, K. Liu et al. [30] highlighted challenges related to machine learning techniques, identifying several factors that could lead to biased results, such as the quality of the sample set. The authors argue that samples of poor quality, with a non-representative size or outdated samples, may yield promising results in experimental settings but might not perform similarly in a real environment [30]. Another critical aspect is the quality of the extracted feature dataset. The efficacy of machine learning approaches heavily relies on the selection of correct features and their extraction methods, particularly dynamic features. Moreover, in addition to the computational costs involved, other studies [35], [36] have indicated that machine learning approaches exhibit weaknesses when dealing with malicious apps that alter their behavior to mislead learning algorithms, which may restricts their applicability in real-world scenarios.

3 EXPERIMENTAL SETUP

This research aims to develop a deeper understanding of the performance of the MAS approach for detecting malware. To this end, in this paper, we **replicate** the study by Bao et al. [5], which advocates for using the MAS approach for malware classification. However, in contrast to the original study [5], we use a dataset of repackaged apps that is an

order of magnitude larger. Accordingly, we investigate the following research questions:

(RQ1) What is the impact of considering a larger and diverse dataset on the accuracy of the MAS approach for malware classification? Answering this question may help shed light on potential generalization issues in previous studies that empirically assess the MAS approach approach to 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? Answering this research question helps clarify whether the similarity between an original app and its repackaged version affects the MAS approach approach's performance in 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? Answering this research question may help identify potential blind spots in the MAS approach approach to malware classification, revealing possible extensions that could improve the detection of specific malware families.

In this section, we describe our study settings. First, we present our procedures to create our datasets (Section 3.1). Then, we describe the data collection and data analysis procedures (Sections 3.2 and 3.3).

3.1 Malware Dataset

To address our research questions, we contribute a dataset designed to meet two primary requirements. First, it should provide a comprehensive and up-to-date selection of Android repackaged apps. By "comprehensive", we mean at least an order of magnitude larger than the dataset used in the original study [5]. Given its comprehensiveness, we expect it to include a diverse range of malware families to ensure representativeness. Second, our dataset should be properly labeled, ensuring each sample includes key attributes such as similarity and malware family. This is particularly necessary to answer research questions RQ2 and RQ3.

3.1.1 Procedures for Building the Dataset

We curate our dataset in three main phases. In the first phase, we use two repositories of repackaged Android apps (RePack [7] and AndroMalPack [37]) to build the dataset we use in our research. RePack was curated using automatic procedures that extract repackaged apps from the Androzoo repository [38]. It comprises 18,073 apps, from which 2,776 are original versions of an app and the remaining ones are repackaged. RePack contains 15,297 pairs of original and repackaged Android apps, many repackaged versions of the same original app may coexist within the RePack dataset—note that all repackaged variants of a given app are derived from the same original version, as confirmed by their matching hash identifier. RePack is the leading dataset used in Android repackaged research [15], even though it only contains packages built until 2018. For this reason, we decided to include samples from the AndroMalPack

dataset collected after 2018 in our research. Unlike RePack, AndroMalPack lacks information about the original apps, leading us to follow an existing heuristic [7] to identify the original versions of its repackaged apps, leading to a sample from the AndroMalPack dataset that contains 1,190 pairs (original/repackaged) of apps, all pairs satisfying our constraint of being collected after 2018. Altogether, our initial dataset contains a total of 16,487 pairs of apps.

In the second phase, we discarded some samples from our initial dataset because, during the execution of our experiments, we encountered recurrent issues related to the instrumentation of the apps using DroidFax [24]. Other problems occurred after the execution of the apps in the Android emulator, while analyzing the apps or their execution logs. More precisely, we encountered failures while instrumenting 919 original apps from our initial dataset, including both RePack and AndroMalPack. After removing these original apps from our dataset, we were left with 5,875 pairs (original/repackaged) of apps. Among these pairs, 430 repackaged apps could not be instrumented. Failures also occurred while analyzing either the original or repackaged version of 586 apps, resulting in a dataset containing 4,742 pairs. Failures at this phase were expected, as some malware samples employ evasion tactics, such as deliberately crashing test apps in simulated environments, to avoid detection [39]. Finally, we could not install five apps in the version of the Android emulator (API level 28) we used in our research. Compared to our experience building our dataset, a more significant percentage of failures has been reported in previous research [5]. Note that we did not apply any filters to increase the representation of certain malware families in our dataset.

Third, we queried the `VirusTotal` repository to identify original versions of apps labeled as malware. Samples with such labels were excluded from our dataset, as the MAS approach assumes that the original version of an app is not malware (otherwise, the repackaged versions might also exhibit malicious behavior). `VirusTotal` is a widely recognized tool that scans software assets, including Android apps, using over 60 antivirus engines [15]. Thus, we excluded 661 samples from our dataset that do not satisfy this constraint.

In the end, we are left with our final dataset (hereafter `LargeDS`) of 4,076 apps which we use in our study. To bring evidence that we were able to reproduce the results of previous research, we also consider in our research a small dataset (`SmallDS`) used in the original study [5]. This is the same dataset referenced in Section 2.2 as (`LargeE`).

3.1.2 Features of the Datasets

We queried the `VirusTotal` repository to find out which repackaged apps in our dataset have indeed been labeled as a malware. According to `VirusTotal`, in the `SmallDS` (102 pairs), 69 of the repackaged apps (67.64%) were identified as malware by at least two security engines. Here, we consider a repackaged version of an app to be malware only if `VirusTotal` reports that at least two security engines identify malicious behavior within the asset. Although this decision aligns with previous research [40], [15], we assess its potential impact on our findings in Section 5. Considering the `LargeDS`, at least two security engines identified 2,895

out of the 4,076 repackaged apps as malware (71.02%). Again, in Section 5, we show that our results remain consistent across three additional scenarios: classifying a repackaged version of an app as malware if at least one, five, or ten `VirusTotal` engines flag it as malicious.

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 on its characteristics and attack strategy—e.g., steal network info (IP, DNS, WiFi), collect phone info, collect user contacts, send/receive SMS, and so on [41]. According to the `avclass2` tool [42], the malware samples in the `SmallDS` come from 17 different families—most of them from the Kuguo (49.27%) and Dowgin (17.39%) families. Our `LargeDS`, besides comprising a large sample of repackaged apps (4,076 in total), contains 116 malware families—most of them from the Gappusin (46.18%) family. Despite being flagged as malicious by at least two security engines, unfortunately `avclass2` tool cannot correctly identify the family of 253 samples in our `LargeDS`.

We also characterize our dataset according to the similarity between the original and repackaged versions of the apps, using the SimiDroid tool [43]. 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 `LargeDS` has an average similarity score of 90.39%, with the following distribution: 87 app pairs have a similarity score below 25%, 49 pairs fall between 25% and 50%, 353 apps between 50% and 75%, and 3,587 apps exceed 75%. The `SmallDS` has an average similarity score of 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 our dataset (`SmallDS` and `LargeDS`), malicious app versions injected 133 distinct methods from sensitive APIs (according to the AppGuard [44] security framework). Malicious code often exploits these APIs to compromise system security and access sensitive data. Table 1 lists the 10 most frequently called methods from sensitive APIs that appear only in the repackaged versions of the apps.

We must highlight that the `LargeDS` samples come from different Android app stores. Most of our repackaged apps come from a non-official Android app store, Anzhi [45]. 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 [23] for data collection. DroidXP allows researchers to compare test case generation tools for malicious app behavior 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.

(Step1) **Instrumentation**: In the first step, we configure DroidXP to instrument all pairs of apps in our datasets (`SmallDS` and `LargeDS`). Here, we instrument both versions of the apps (as APK files) to collect relevant information during their execution. Under the hood, DroidXP leverages DroidFax 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.

(Step2) **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 repackaged). We execute the apps via DroidBot [11], mainly because the original research we replicate here reports that DroidBot leads to the best accuracy of the MAS approach for malware identification. Since previous studies suggest that DroidBot's coverage nearly reaches its maximum within one minute [5], we run each app for three minutes. To mitigate the randomness inherent in test case generation tools, we repeat this process three times. 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.

(Step3) **Data Collection**: After the execution of the instrumented apps, once again, DroidXP leverages DroidFax, this time to collect all relevant information (such as calls to sensitive APIs, test coverage metrics, and so on). We use this information to analyze the performance of the MAS approach for detecting malicious behavior.

3.3 Data Analysis Procedures

We consider that the MAS approach builds a sandbox that labels a repackaged version of an app as malware if there is at least one call to a sensitive API 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 malware. The set of sensitive APIs we use was defined in the AppGuard framework [44], which was based on the mapping from sensitive APIs to permissions proposed by Song et al. [46]. We triangulate the results of the MAS approach classification 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 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 malware and, according to `VirusTotal`, at most one security engine labels the asset as a malware.

Method of Sensitive API	Occurrences
android.telephony.TelephonyManager: int getPhoneType()	311
android.telephony.TelephonyManager: java.lang.String getNetworkOperatorName()	297
android.location.LocationManager: java.lang.String getBestProvider(android.location.Criteria,boolean)	292
android.telephony.TelephonyManager: int getSimState()	284
java.lang.reflect.Field: java.lang.Object get(java.lang.Object)	277
android.net.NetworkInfo: java.lang.String getTypeName()	271
android.database.sqlite.SQLiteDatabase: android.database.Cursor query(java.lang.String,java.lang.String[],...,...,...,...,...)	270
java.lang.reflect.Field: int getInt(java.lang.Object)	250
android.net.wifi.WifiInfo: java.lang.String getMacAddress()	238
android.telephony.TelephonyManager: java.lang.String getNetworkOperator()	237

TABLE 1: Sensitive APIs that frequently appear in the repackaged versions of the apps.

Study Feature	Original Study	Replication Study
Dataset	102 pairs of samples	4,076 pairs of samples
Execution time	One minute	Three minutes
Number of executions	Single execution	Three executions
Metrics	Malware prevalence	Precision, Recall, and F1-score
Test case generation tool	Six different tools (DroidBot with best performance)	DroidBot only

TABLE 2: Characterization of this replication study

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

In Section 4 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, using both datasets—i.e., the `SmallDS` with 102 pairs of apps and `LargeDS` with 4,076 pairs. We use the Spearman Correlation [47] method and Logistic Regression [48] 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 classify an asset as malware correctly. 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.

Table 2 highlights the differences between the original study [5] and our replication study. In the best-case scenario, where no re-executions are required, our experiment would take at least 611 hours. In contrast, under the best conditions, the original experiment's execution would last 60 hours. This difference is one of the reasons we focus our research on DroidBot, the test case generation tool that demonstrated the best performance in the original study.

4 RESULTS

In this section, we detail the findings of our study. We remind the reader that this replication study's primary goal is to better understand the strengths and limitations of the MAS approach for malware detection by replicating the work of Bao et al., using DroidBot as the test case generation tool. We explore the results of our research using two datasets: the `SmallDS` (102 app pairs), and `LargeDS` (4,076 pairs).

4.1 Exploratory Data Analysis of Accuracy

SmallDS. Considering the `SmallDS` (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. reported [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 could reproduce the findings of the original study using our implementation settings of the MAS approach.

> **Finding 1**. We were able to reproduce the results of existing research using our implementation of the MAS approach, achieving a malware classification in the `SmallDS` close to what has been reported in previous studies.

In the original study [5], the authors assume that all repackaged versions are malware and contain a malicious code. 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 original study. As we mentioned, in

TABLE 3: Accuracy of the MAS approach in both datasets.

Dataset	TP	FP	FN	Precision	Recall	F_1
SmallDS (102)	63	6	7	0.91	0.90	0.90
LargeDS (4,076)	1,175	220	1,720	0.84	0.40	0.54

this paper we take advantage of VirusTotal to label our dataset and build a ground truth: In our datasets, we classify a repackaged version of an app as malware if, according to our VirusTotal query results, at least two security engines identify malicious behavior in the asset. This decision aligns with existing recommendations [40], [15]). The first row of Table 3 shows that the MAS approach achieves an accuracy of 0.89 when considering the SmallDS. Nonetheless, the MAS approach fails to classify seven assets as malware on the SmallDS correctly (FN column, first row of Table 3), and wrongly labeled the repackaged version of six apps as malware (FP column).

LargeDS. Surprisingly, considering our complete dataset (4,076 apps), the MAS approach labels a total of 1,395 repackaged apps as malware (34.22% 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 SmallDS (see the second row of Table 3): F_1 dropping from 0.89 to 0.54. This result indicates that, when considering a large dataset, the accuracy of the MAS approach using DroidBot drops significantly.

💡 **Finding 2**. The MAS approach for malware detection leads to a substantially lower performance on the LargeDS (4,076 pairs of apps), dropping F1-score from 0.89 to 0.54 in comparison to what we observed in the SmallDS.

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 3. The negative performance of the MAS approach in the LargeDS encouraged us to endorse efforts to identify potential reasons for this phenomenon and motivated us to explore the research questions RQ2 and RQ3.

4.2 Assessment Based on Similarity Score

Figure 1 shows the Similarity Score distribution over the LargeDS we use in our research. Recall that the Similarity Score measures how similar an app's original and repackaged versions are. The complete dataset averages a Similarity Score of 0.90 (with a median of 0.98 and standard deviation of 0.18).

In this section we investigate how the Similarity Score influences the accuracy of the MAS approach—which might help us understand if it might explain the low small performance of the MAS approach in the LargeDS. To this end, we leverage Logistic Regression to quantify the relationship between Similarity Score and F1-score. This analysis excludes instances of true negatives (i.e., cases where the repackaged version is benign according to VirusTotal and the MAS
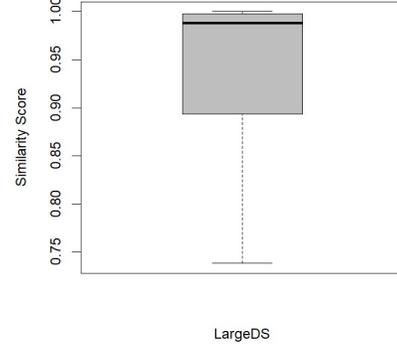

Fig. 1: Similarity Score of the malware samples in the LargeDS. The boxplots in the figure do not show outliers.

approach correctly labels it 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 logistic regression results suggest that we should reject our null hypothesis (p-value $= 2.22 \cdot 10^{-16}$). This finding indicates that the accuracy of the MAS approach on LargeDS is influenced by the similarity between the original and repackaged versions of an app.

💡 **Finding 3**. There is an association between the Similarity Score and the MAS approach performance, which means that the similarity between the original and repackaged versions of an app can explain the performance of the MAS approach for malware classification.

To clarify the association between Similarity Score and accuracy, we use the *K-Means* algorithm to split the LargeDS into ten clusters—according to the Similarity Score. We then estimate the percentage of correct classifications for each cluster, as shown in Table 4. Note that the MAS approach achieves the highest percentage of correct classification (77.35%) for the second cluster (cId = 2), which presents an average Similarity Score of (0.56). Nonetheless, the cluster cId = 10, with a larger number of samples (1,302) and Similarity Score (0.99), presents a percentage of correct classifications of 26.5%. We can observe that as the average similarity rate decreases, there is a tendency toward greater accuracy in the MAS approach. Hence, the average similarity score could explain the poor performance of the MAS approach on the LargeDS, especially considering that most samples exhibit a high average similarity of 0.99.

TABLE 4: Characteristics of the clusters. Note there is a 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	%
1	0.42	42	30	71.42
2	0.56	181	140	77.35
3	0.68	131	98	74.81
4	0.80	170	104	61.18
5	0.88	263	83	31.56
6	0.91	236	129	54.66
7	0.95	167	51	30.54
8	0.97	150	67	44.67
9	0.98	421	112	26.60
10	0.99	1302	345	26.50

4.3 Assessment Based on Malware Family

As we discussed in the previous section, the similarity assessment partially explains the low performance of the MAS approach on the `LargeDS`. Since the `LargeDS` covers a wide range of malware families, we investigate the hypothesis that the diversity of malware families in the `LargeDS` also contributes to the poor performance of the MAS approach on the `LargeDS`. Indeed, in the `LargeDS`, we identified a total of 116 malware families, though the most frequent ones are *gappusin* (1,337 samples), *revmob* (207 samples), *dowgin* (183 samples) and *airpush* (120 samples). Together, they account for 63.79% of the repackaged apps in our `LargeDS` labeled as malware according to `VirusTotal`.

This family distribution in the `LargeDS` is different from the family distribution in the `SmallDS` (used in the original study)—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 in the `SmallDS` for which `VirusTotal` labels as malware. Most important, in the `SmallDS`, there is just one sample from the *gappusin* family and no sample from *revmob* family, two of the most frequent families in our `LargeDS`. This observation leads us to the question: how does the MAS approach perform when considering only samples from the *gappusin* and *revmob* families?

The confusion matrix of Table 5 summarizes the accuracy assessment of the MAS approach considering only the *gappusin* and *revmob* samples in the `LargeDS`. To make clear, `VirusTotal` classifies as malware all repackaged versions in the *gappusin* and *revmob* family. It is worth noting that the MAS approach failed to classify correctly 1,170 (87.5%) samples of *gappusin* as malware. Similarly, 92 samples (44.44%) from *revmob* were not classified as malware. Furthermore, if we exclude the *gappusin* and *revmob* samples from the `LargeDS`, the recall of the MAS approach increases to 0.72, which, although improved, remains relatively low compared to the original studies.

TABLE 5: Confusion matrix of the MAS approach when considering only the samples from the *gappusin* and *revmob* family in the `LargeDS`.

Actual Condition	Predicted Condition	
	Benign	Malware
Benign (0)	TN (0)	FP (0)
Gappusin (1,337)	FN (1,170)	TP (167)
Revmob (207)	FN (92)	TP (115)

Finding 4. The MAS approach fails to correctly identify 87.50% of the samples from the *gappusin* family and 44.44% of the samples from the *revmob* as malware. Just like the Similarity Score, the presence of some malware families with a high false negative rate also influences the low recall of the MAS approach in the `LargeDS`

We further analyze the samples from the *gappusin* and *revmob* malware families in our dataset, given their relevance to the negative results presented in our paper. First, we examined the Similarity Score of the samples. Figure 2 shows a histogram of the Similarity Score for both families. Most repackaged versions are similar to the original ones, with an average Similarity Score of 0.94, a median of 0.99, and a standard deviation (SD) of 0.16 for the *gappusin* family. For the *revmob* family, the average Similarity Score is 0.81, the median is 0.91, and the SD is 0.26.

We also reverse-engineered samples from both families. Due to the significant effort required for reverse engineering, we limited our analysis to a sample of 30 *gappusin* and 30 *revmob* malware samples, using the `SimiDroid`[3], `apktool`[4], and `smali2java`[5] tools. Considering this sample, the median Similarity Score is 0.99 and 0.90 for the *gappusin* and *revmob* families, respectively. Table 6 and Table 7 summarize the outputs of `SimiDroid` for these samples.
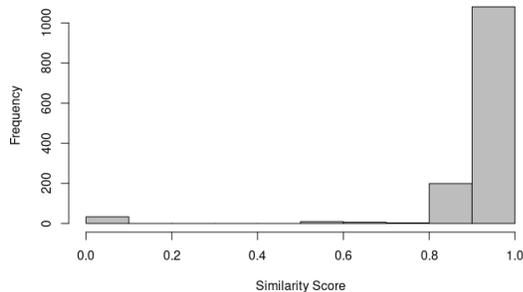
Regarding the *gappusin* malware, the similarity assessment of this sample of 30 apps reveals a few modification patterns when comparing the original and the repackaged versions. First, no instance in this *gappusin* sample dataset modifies the Android Manifest file to require additional permissions. In most 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 behavior of the benign version, so that it can download a different version of the `data.apk` asset. Figure 3 shows the code pattern of the `onReceive` method present in the samples. This modification typically uses a new method (`public void a(Context)`) in the repackaged versions, often introduced into the same class (`AdReceiver`). Since there are no additional calls to sensitive APIs, the MAS approach fails to correctly label
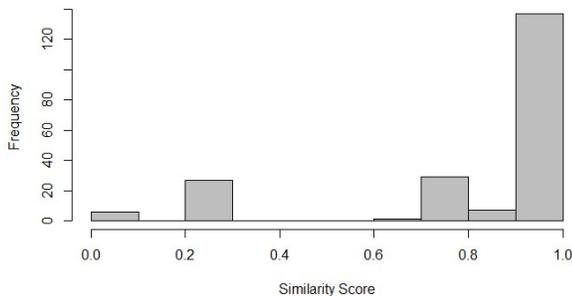
3. https://github.com/lilicoding/SimiDroid
4. https://ibotpeaches.github.io/Apktool/
5. https://github.com/AlexeySoshin/smali2java

the *gappusin* samples. This limitation holds regardless of our experimental choices, such as using the DroidBot tool (instead of more recent test case generation tools) or running the samples for three minutes.

(a) Similarity Score for the samples in the *gappusin* family.

(b) Similarity Score for the samples in the *revmob* family.

Fig. 2: Histogram of the Similarity Score for the samples in the *gappusin* and *revmob* families.

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 uses 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 a strategy for antivirus evasion. For instance, although some usages of the class `DexClassLoader` might be legitimate, it allows specific types of attack based on dynamic code injection [49]. 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), again,

regardless of the decisions we follow in our experiment. Listing 4 shows an example of code pattern frequently removed from the repackaged versions from the *gappusin* family.

TABLE 6: 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

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. It acts as a downloader for further malicious app [13]—which reduces the ability of the MAS approach to classify a sample as a malware correctly. Both versions (original/repackaged) from the *gappusin* family have the same behavior for showing advertisements to the user, however, the repackaged version has additional call sites to the advertisement API and the advertisement sources are different.

Similar to the approach used for *gappusin* samples, we also reverse-engineered a random selection of 30 samples from the *revmob* family that were not detected by the MAS approach. As with the *gappusin* family samples, no instance from the *revmob* family modifies the Manifest file or inserts extra calls to sensitive APIs, making it harder for the MAS approach to label the samples as malware correctly. However, our reverse engineering reveals that all apps store a file with the extension ".so" (Shared Object files) in their lib directory. These files are dynamic libraries containing native code written in C or C++, and are often used by apps for performance reasons, when resource-intensive tasks need to be performed [50].

Unfortunately, creating malicious repackaged apps using ".so" files is also possible, as they can be replaced by a

```
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();
  }
}
```

Fig. 3: Method introduced in 29 out of 30 *gappusin* malware we randomly selected from the `LargeDS`.

```
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();
  }
}
```

Fig. 4: Example of method that is typically removed from the repackaged apps of the *gappusin* family.

version containing harmful code [51]. The Shared Object files also allow for attacks based on dynamic code injection [49], considering that Android apps can use methods like `System.loadLibrary()` or `System.load()` to download malicious ".so" files from a remote server. Once on the device, malicious apps can use these files to interface with Java code in Android apps, via the Java Native Interface (JNI), performing malicious operations on low-level code and bypassing security mechanisms, like the MAS approach.

Our assessment confirms that all *revmob* samples contain Java code that loads a native library. In particular, to load these libraries, the samples use the `loadLibrary` method of the `System` class, which is called in the static constructor of the `mainActivity` class. The `loadLibrary` method takes "game" as an argument, and the code automatically searches the default lib directory for the `.so` file named (`lib+argument`). The "lib" directory contains the `libgame.so` file in all samples. Figure 5 presents the code pattern of the `mainActivity` class found in the samples from *revmob* family.

The `libgame.so` file contains compiled code written in C or C++, which is loaded into memory and linked to the apps at runtime. Although machine code is difficult to analyze, all the files include the `JNI_OnLoad` function, which the JNI implementation automatically uses to link

```
public class PZPlayer extends Cocos2dxActivity {
    // ...
    System.loadLibrary("game");
    // ...
}
```

Fig. 5: Thie code links this java file into libgame shared library

Java methods and native functions. When we analyzed the ".so" file, we found that they all differ in size and content between the original and repackaged apps. It is possible that changes of interest occurred in the native `libgame.so` file and have gone unnoticed by the MAS approach. Again, since the MAS approach only considers differences in calls to sensitive APIs, it is unlikely to correctly classify these samples using other test case generation tools or by extending the execution time during its exploratory phase.

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.

TABLE 7: Summary of the outputs of the `SimiDroid` tool for the sample of 30 *revmob* malware. (IM) Identical Methods, (SM) Similar Methods, (NM) New Methods, and (DM) Deleted Methods.

Hash	Similarity Score	IM	SM	NM	DM
14BBE2	0.9940	3348	6	532	14
BFEF74	0.9940	3348	6	532	14
A3FACA	0.7918	2667	80	112	1 621
10F22D	0.9940	3348	6	532	14
50193A	0.9940	3348	6	532	14
5A7536	0.9940	3348	6	532	14
BCC0DB	0.7918	2667	80	112	1 621
E866CB	0.9940	3348	6	532	14
CDD316	0.9940	3348	6	532	14
DF39F6	0.7918	2667	80	112	1 621
3FFAFF	0.9121	3072	184	628	112
C8C63D	0.9940	3348	6	532	14
48C562	0.9121	3072	184	628	112
D27F26	0.7918	2667	80	112	1 621
F4BBEC	0.9121	3072	184	628	112
BCF14C	0.9127	3074	182	628	112
7FBF11	0.7918	2667	80	112	1 621
9D35D4	0.7918	2667	80	112	1 621
D1B27E	0.9940	3348	6	532	14
94DD4B	0.9940	3348	6	532	14
2D217E	0.7918	2667	80	1121	621
66F167	0.7918	2667	80	1121	621
155D4A	0.9940	3348	6	532	14
8CB780	0.9127	3074	82	628	112
C251FA	0.9940	3348	6	532	14
40487B	0.7918	2667	80	1121	621
F29692	0.9940	3348	6	532	14
0E3679	0.9127	3074	182	628	112
7A4F31	0.9121	3072	184	628	112
BB3EDE	0.7105	2393	256	1217	719

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 (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 `SmallDS` 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 0.54 in the presence of our `LargeDS` (4,076 pairs of original and repackaged versions of Android apps).

- **Similarity Analysis (RQ2).** Our results bring evidence about the 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 is relevant for explaining the performance of the MAS approach to classify certain repackaged versions of an app as malware.

- **Malware Family Analysis (RQ3).** The results indicate that some families are responsible for the largest number of false negatives in the complete dataset. We specifically further investigate the *gappusin* and *revmob* families—a particular type of Adware, designed to display advertisements while an app is running automatically. After reverse engineering a sample of 60 malware apps from *gappusin* and *revmob* family, 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* and *revmob* families in the Android malware landscape accounts for the poor performance of the MAS approach in malware classification on the large dataset.

5.2 Implications

Contrasting to previous research works [5], [6], [9], 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`—a common practice in the malware identification research. This decision allowed us to explore the MAS approach performance using well-known accuracy metrics (i.e., precision, recall, and F_1 score). Contrasting with previous studies that 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 `SmallDS` ($F_1 = 0.89$), in the presence of a large dataset the MAS approach accuracy drops significantly ($F_1 = 0.54$).

We also reveal that some families in the `LargeDS` are 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 sophisticated techniques to differentiate between benign and malicious apps. In particular, since our reverse engineering results suggest that *gappusin* and *revmob*—two recurrent malware families— use the network to download new assets, new approaches might benefit from monitoring not only calls to sensitive APIs but also network traffic, as well as mining sensitive calls to native APIs embedded in `so` files. The versatility of the Java Native Interface (JNI) has introduced challenges. Malware authors increasingly use the native layer to hide malicious code, making both static and dynamic analysis more difficult. The current state of the art in sandbox mining overlooks native calls.

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 on the MAS approach for malware classification, 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 substantially superior to its performance on our `LargeDS` (4,076 pairs of apps).

We contacted the authors of the Bao et al. original 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 `SmallDS`. We believe our results in the `LargeDS` generalize better than previous research work, since we have a more comprehensive collection of malware with different families and degrees of similarity. Nonetheless, our research focuses only on Android repackaged malware. Thus, we cannot generalize our findings to malware that targets other platforms or uses different approaches to instantiate a malicious asset. Besides that, repackaging is a recurrent approach for implementing Android malware.

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). In the exploratory phase, the MAS approach assumes the existence of a benign original version of a given app. We also query `VirusTotal` to confirm this assumption, and found that the original version of seven (out 102) apps in the `SmallDS` 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 [40], and a dataset that is consistent on a given date may not remain consistent 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 `LargeDS` 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). We computed the accuracy results using precision and recall based on these metrics. In a preliminary study, we investigated whether or not the MAS approach would classify an original version of an app as 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 malware. Also, we label a repackaged version of an app as malware only if `VirusTotal` reports that at least two engines detect suspicious behavior in that asset. This decision might be viewed as either a weak or strong constraint and could raise concerns about construct validity. However, when we relax this constraint and label an asset as malware whenever at least one engine detects suspicious behavior, precision improves to 0.85, but recall drops to 0.39. Overall, the accuracy of the MAS approach remains almost unchanged ($F_1 = 0.53$)—still significantly lower than the precision of the

MAS approach for `SmallDS`. We also evaluated accuracy by considering an asset as malware when at least five or ten `VirusTotal` security engines flagged it. As shown in Table 8, the results did not diverge significantly from what we have reported in this paper.

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 replicates previous research works [5], [9]. The study utilizes a more diverse dataset compared to those used in previous research, with the aim of providing a more comprehensive evaluation of the approach. To our surprise, compared to published results, the performance of the MAS approach drops significantly for our comprehensive dataset (F_1 score reduces from 0.89 in previous papers to 0.54 here). This result is partially explained by the high prevalence of specific malware families (named *gappusin* and *revmob*), whose samples are incorrectly classified by the MAS approach. We also report the results of a reverse engineering effort, whose goal was to understand the characteristics of the *gappusin* and *revmob* 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. Similarly, the *revmob* family does not include any additional calls to sensitive resources; however, it often uses JNI to interact with native code, which can be used to perform malicious operations at a low level, compromising the effectiveness of the MAS approach for malware identification. These negative results highlight the current limitations of the MAS approach for malware classification and suggest the need for further research to integrate the MAS approach with other techniques for more effective malware identification.

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, "Statista," https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, accessed: 2023-08-25.

[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. 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

TABLE 8: Accuracy of the MAS approach at `LargeDS` (4,076 pairs) based on engines.

Engine(s)	TP	FP	FN	Precision	Recall	F_1
At least 01	1,222	220	1,900	0.85	0.39	0.53
At least 02	1,175	220	1,720	0.84	0.40	0.54
At least 05	1,087	220	1,578	0.83	0.40	0.54
At least 10	1,002	220	1,469	0.81	0.40	0.54

[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] google, "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] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket

[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] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] S. Seraj, M. Pavlidis, and N. Polatidis, "Trojandroid: Android malware detection for trojan discovery using convolutional neural networks," in *Engineering Applications of Neural Networks - 23rd International Conference, EAAAI/EANN 2022, Chersonissos, Crete, Greece, June 17-20, 2022, Proceedings*, ser. Communications in Computer and Information Science, L. Iliadis, C. Jayne, A. Tefas, and E. Pimenidis, Eds., vol. 1600. Springer, 2022, pp. 203–212. [Online]. Available: https://doi.org/10.1007/978-3-031-08223-8_17

[26] A. Lekssays, B. Falah, and S. Abufardeh, "A novel approach for android malware detection and classification using convolutional neural networks," in *Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020*, M. van Sinderen, H. Fill, and L. A. Maciaszek, Eds. ScitePress, 2020, pp. 606–614. [Online]. Available: https://doi.org/10.5220/0009822906060614

[27] L. Wei, W. Luo, J. Weng, Y. Zhong, X. Zhang, and Z. Yan, "Machine learning-based malicious application detection of android," *IEEE Access*, vol. 5, pp. 25 591–25 601, 2017. [Online]. Available: https://doi.org/10.1109/ACCESS.2017.2771470

[28] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, E. Bertino and R. S. Sandhu, Eds. ACM, 2012, pp. 317–326. [Online]. Available: https://doi.org/10.1145/2133601.2133640

[29] M. Choudhary and B. Kishore, "Haamd: Hybrid analysis for android malware detection," in *2018 International Conference on Computer Communication and Informatics (ICCCI)*. IEEE, 2018, pp. 1–4.

[30] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124 579–124 607, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3006143

[31] 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, pp. 76:1–76:41, 2017. [Online]. Available: https://doi.org/10.1145/3017427

[32] M. Odusami, O. Abayomi-Alli, S. Misra, O. Shobayo, R. Damasevicius, and R. Maskeliunas, "Android malware detection: A survey," in *Applied Informatics - First International Conference, ICAI 2018, Bogotá, Colombia, November 1-3, 2018, Proceedings*, ser. Communications in Computer and Information Science, H. Florez, C. Diaz, and J. Chavarriaga, Eds., vol. 942. Springer, 2018, pp. 255–266. [Online]. Available: https://doi.org/10.1007/978-3-030-01535-0_19

[33] V. Sihag, M. Vardhan, and P. Singh, "A survey of android application and malware hardening," *Comput. Sci. Rev.*, vol. 39, p. 100365, 2021. [Online]. Available: https://doi.org/10.1016/j.cosrev.2021.100365

[34] M. T. Ahvanooey, Q. Li, M. Rabbani, and A. R. Rajput, "A survey on smartphones security: Software vulnerabilities, malware, and attacks," *CoRR*, vol. abs/2001.09406, 2020. [Online]. Available: https://arxiv.org/abs/2001.09406

[35] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, machine learning can be more secure! A case study on android malware detection," *IEEE Trans. Dependable Secur. Comput.*, vol. 16, no. 4, pp. 711–724, 2019. [Online]. Available: https://doi.org/10.1109/TDSC.2017.2700270

[36] J. Gardiner and S. Nagaraja, "On the security of machine learning in malware c&c detection: A survey," *ACM Comput. Surv.*, vol. 49, no. 3, pp. 59:1–59:39, 2016. [Online]. Available: https://doi.org/10.1145/3003816

[37] H. Rafiq, N. Aslam, M. Aleem, B. Issac, and R. H. Randhawa, "Andromalpack: enhancing the ml-based malware classification by detection and removal of repacked apps for android systems," *Scientific Reports*, vol. 12, no. 1, p. 19534, 2022.

[38] 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

[39] O. Alrawi, M. Y. Wong, A. Avgetidis, K. Valakuzhy, B. V. Adjibi, K. Karakatsanis, M. Ahamad, D. M. Blough, F. Monrose, and M. Antonakakis, "Sok: An essential guide for using malware sandboxes in security applications: Challenges, pitfalls, and lessons learned," *CoRR*, vol. abs/2403.16304, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.16304

[40] 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 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2361–2378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zhu

[41] 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

[42] S. Sebastián and J. Caballero, "Avclass2: Massive malware tag extraction from AV labels," in *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 2020, pp. 42–53. [Online]. Available: https://doi.org/10.1145/3427228.3427261

[43] 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

[44] 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

[45] none, "anzhi," http://www.anzhi.com/, accessed: 2021-02-15.

[46] 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

[47] 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

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

[49] 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, Los Angeles, CA, USA, December 7-11, 2015*. ACM, 2015, pp. 201–210. [Online]. Available: https://doi.org/10.1145/2818000.2818042

[50] A. Ruggia, A. Possemato, S. Dambra, A. Merlo, S. Aonzo, and D. Balzarotti, "The dark side of native code on android," *Authorea Preprints*, 2023.

[51] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan, "On tracking information flows through JNI in android applications," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 180–191. [Online]. Available: https://doi.org/10.1109/DSN.2014.30