Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

12-2018

Towards mining comprehensive Android sandboxes

Tien-Duy B. LE
Singapore Management University, btdle.2012@smu.edu.sg

Lingfeng BAO
Zhejiang University

David LO
Singapore Management University, davidlo@smu.edu.sg

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

Li LI
Monash University

DOI: https://doi.org/10.1109/ICECCS2018.2018.00014

Citation

LE, Tien-Duy B.; BAO, Lingfeng; LO, David; GAO, Debin; and LI, Li. Towards mining comprehensive Android sandboxes. (2018). *ICECCS 2018: 23rd International Conference on Engineering of Complex Computer Systems: Proceedings : 12-14 December, Melbourne, Australia*. 51-60. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/4289

Towards Mining Comprehensive Android Sandboxes

Tien-Duy B. Le*, Lingfeng Bao†, David Lo*, Debin Gao*, Li Li‡
*School of Information Systems, Singapore Management University, Singapore
†School of Computer and Computing Science, Zhejiang University City College, China
‡Faculty of Information Technology, Monash University, Australia
{btdle, davidlo, dbgao}@smu.edu.sg, lingfengbao@zju.edu.cn, li.li@monash.edu

***Abstract*—Android is the most widely used mobile operating system with billions of users and devices. The popularity of Android apps have enticed malware writers to target them. Recently, Jamrozik et al. proposed an approach, named Boxmate, to mine sandboxes to protect Android users from malicious behaviors. In a nutshell, Boxmate analyzes the execution of an app, and collects a list of sensitive APIs that are invoked by that app in a monitoring phase. Then, it constructs a sandbox that can restrict accesses to sensitive APIs not called by the app. In such a way, malicious behaviors that are not observed in the monitoring phase – occurring, for example, due to malicious code injection during an attack – can be prevented. Nevertheless, Boxmate only focuses on a specific API type (i.e., sensitive APIs); it also ignores parameter values of many API methods and requested permissions during the execution of a target app. As a result, Boxmate is not able to detect malicious behaviors in many cases.**

In this work, we address the limitation of Jamrozik et al.'s work by considering input parameters of many different types of API methods for mining a more comprehensive sandbox. Given a benign app, we first extract a list of Android permissions that the app may request during its execution. Next, we leverage an automated test case generation tool, named Droidbot, to generate a rich set of GUI test cases for exploring behaviors of the app. During the execution of these test cases, we analyze the execution of four different types of API methods. Furthermore, we record input parameters to these API methods, and classify those into four different categories. We leverage the collected parameter values, and the list of requested permissions to create a sandbox that can protect users from malicious behaviors. Our experiments on 25 pairs of real benign and malicious apps show that our approach is more effective than the coarse- and fine-grained variants of Boxmate by 267.37% and 81.64% in terms of F-measure respectively.

***Index Terms*—Mining Sandboxes, Malicious Behavior Detection, Android Security**

I. Introduction

Nowadays, Android is the most widely used mobile operating system. Nevertheless, Android users easily become targets of malwares and attackers. Truong et al. highlighted that 0.25% of Android devices are affected by malware [1], which constitutes a considerable large number of infected devices. Recently, Jamrozik et al. [2] proposed an approach, named Boxmate, to mine sandboxes for Android apps. Boxmate prevents the executions of suspicious behaviors that have not been observed from the executions of an app in a monitoring phase by automatically constructing sandboxes. In particular, Boxmate utilizes an automated test case generation tool to generate GUI test cases for exploring behaviors of an app. During the execution of these test cases, Boxmate records occurrences of sensitive API methods (e.g., methods that take photos or request access to location information, etc.). It also considers input parameters of methods in Android's *ContentResolvers* classes. Boxmate leverages the above collected information during the execution of apps to form sandboxes. The sandboxes are then used to detect suspicious behaviors that may correspond to the executions of malicious code injected to the app.

In this work, to create more effective sandboxes for malicious behavior detection, we propose a new approach that takes into account various types of APIs and values of parameters that are input to them. Our goal is to construct more accurate sandboxes that can distinguish malicious and benign activities during the execution of Android apps. In particular, given a benign Android app, our approach first extracts a list of requested Android permissions. Next, we leverage an automated test case generation tool to generate a rich set of GUI test cases to explore behaviors of the benign Android app. During the executions of these test cases, our approach analyzes four different types of APIs: *sensitive*, *reflection*, *BroadcastReceiver*, and *SharedPreferences* APIs, which can be exploited by malware or attackers to take control of mobile devices or stole users' data. We also record input parameters of these APIs and classify them into four different categories: *resource*, *reflections*, *SharedPreferences*, and *ContentResolvers*. We utilize recorded parameters and information of requested permissions to form a sandbox that can distinguish malicious and benign behaviors of the given app.

Compared to Boxmate [2], our approach not only records string parameters of *ContentResolvers* but also considers input strings of many other APIs. Additionally, we propose a strategy to detect anomalous values given a set of training strings by leveraging Singular-Value Decomposition technique for dimension reduction and One-class Support Vector Machine for anomaly detection. Compared to exact string matching strategy used by Boxmate, our proposed strategy avoids false alarms but still detects many anomalous strings.

We evaluate our sandbox mining approach on 25 pairs of real benign and malicious Android app pairs (BM pairs). These BM pairs are filtered and selected from a repackaged Android app dataset from Androzoo [3], [4]. All of the malicious apps in the BM pairs that we pick have malicious code (i.e., adware, trojan, spyware, etc.) grafted to the corresponding benign app

Lingfeng Bao is a join first author. He was affiliated with Singapore Management University, Singapore when this work was performed.

in the pair. They thus simulate attacks that can be made to a benign app. Our experiments show that our approach achieves Precision, Recall, and F-measure of 83.33%, 80.00%, and 81.63%, respectively. Furthermore, our approach outperforms two variants of Boxmate [2], which create sandboxes in a coarse-grained and fine-grained manner respectively, in terms of F-measure by 267.37% and 81.64%.

The contributions of our work are highlighted as follows:

1) We propose a sandbox inference approach that comprehensively considers parameters and various types of APIs that Android apps invoke during runtime.
2) We propose a statistical approach based on Singular-Value Decomposition and One-class Support Vector Machine to detect anomalous values given a set of normal string values. Our proposed strategy overcomes the weakness of exact string matching employed by Jamrozik et al. which potentially causes high false alarm rates.
3) We evaluate our proposed approach with 25 pairs of benign and malicious Android apps that are selected from a repackaged Android app dataset from Androzoo [3], [4]. Our results indicate that our approach is more effective than the two baselines in terms of F-measure by 267.37% and 81.64%, respectively.

The rest of our paper is organized as follows: Section II briefly describes background materials. Next, we present our sandbox mining approach in Section III. Then, we discuss our empirical evaluation and findings in Section IV. Section V highlights related work. Finally, we conclude the paper and discuss future work in Section VI.

II. Background

A. Android

Currently, Java is the main programming language to write Android applications. However, Android apps are significantly different from common Java programs. Android apps have no main methods but many entry points that are methods implicitly called by the Android framework. Android framework is responsible for managing the life cycle of all components in an app. An Android app can have four kinds of components, i.e., *Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*, which are corresponding to the top-level abstractions of user interface, background service, data storage, and response to broadcasts, respectively. *Intents* are the inter-component communication (ICC) mechanism in Android. By the difference on whether targeted component is known or not, ICCs can be divided into two categories, i.e., explicit and implicit.

Private and security-sensitive data (e.g., contacts, locations) in Android platform is protected through a permission mechanism. To access a particular sensitive data, an app must call certain APIs after obtaining a suitable permission. These permissions in an app can be declared explicitly in a config file `manifest.xml` or authorized by a user when the app is executing. These sensitive APIs often include operations that are security-critical as they may lead to private data leakage. In this study, we consider the set of sensitive APIs is defined in the AppGuard privacy-control framework [5], which contains a total of 97 APIs.

B. Sandboxing

A sandbox is a security mechanism for separating running programs, which is often used to run untested or untrusted programs. It uses security policies to restrict the resources (e.g., network, disk, etc.) that a program accesses. Android apps run on a VM (Virtual Machine), and are completely isolated from another due to the permissions Android gives each app. This VM guarded by permissions functions like a "sandbox". Unfortunately, Android default permissions are often too coarse-grained. Moreover, Android developers often request more permissions than their apps actually require – this causes the issue of *overprivileged* apps. Felt et al. reported that 33% of Android apps were *overprivileged* [6]. Mining sandboxes for Android is firstly proposed by Jamrozik et al. They explore the behavior of the app under test by an automated test case generation tool and construct a sandbox based on the sensitive APIs identified during testing. The mined sandboxes can detect and prevent unexpected changes in app behaviors.

Jamrozik et al. [2] propose Boxmate that constructs Android sandboxes by analyzing behaviors of benign apps. Boxmate is capable of intercepting hidden attacks, backdoors, and exploited vulnerabilities in Android apps. In Boxmate's approach, there are two main phases: monitoring and deployment phase. The goal of the monitoring phase is to construct a sandbox by analyzing behaviors of a benign app. Boxmate employs a test case generation tool, named Droidmate, to explore the behaviors of the app under test. Droidmate can generate a rich set of test cases, which contain a sequence of GUI events. During the execution of these test cases, Boxmate records invocations of sensitive API calls which are collectively used to construct a sandbox. In the deployment phase, Boxmate deploys the constructed sandbox on an Android app in order to block and raise warnings to the users about anomalous activities of the app that were not observed in the monitoring phase.

In their experiment, Jamrozik et al. evaluated the constructed sandboxes considering two access control levels, i.e., *per-app* and *per-event* access control. With *per-app* access control, the sandboxes are built on the set of sensitive APIs identified by the test case generation tool and check whether there exist different sensitive APIs in deployment phase. The *per-app* access control allows for achieving a quick saturation of sensitive APIs in the monitoring phase while alleviating few false alarms in the deployment phase. However, it may be too coarse to prevent some attacks, e.g., the sensitive APIs invoked by some malicious behavior are exactly the same as those invoked by the normal behavior. Thus, Boxmate implements *per-event* access control, which is a more fine-grained access control policy. For this setting, in the monitoring phase, Boxmate records pairs of sensitive API and the event that triggers it; in the deployment phase, upon invocation of each sensitive API triggered by a GUI event, Boxmate checks

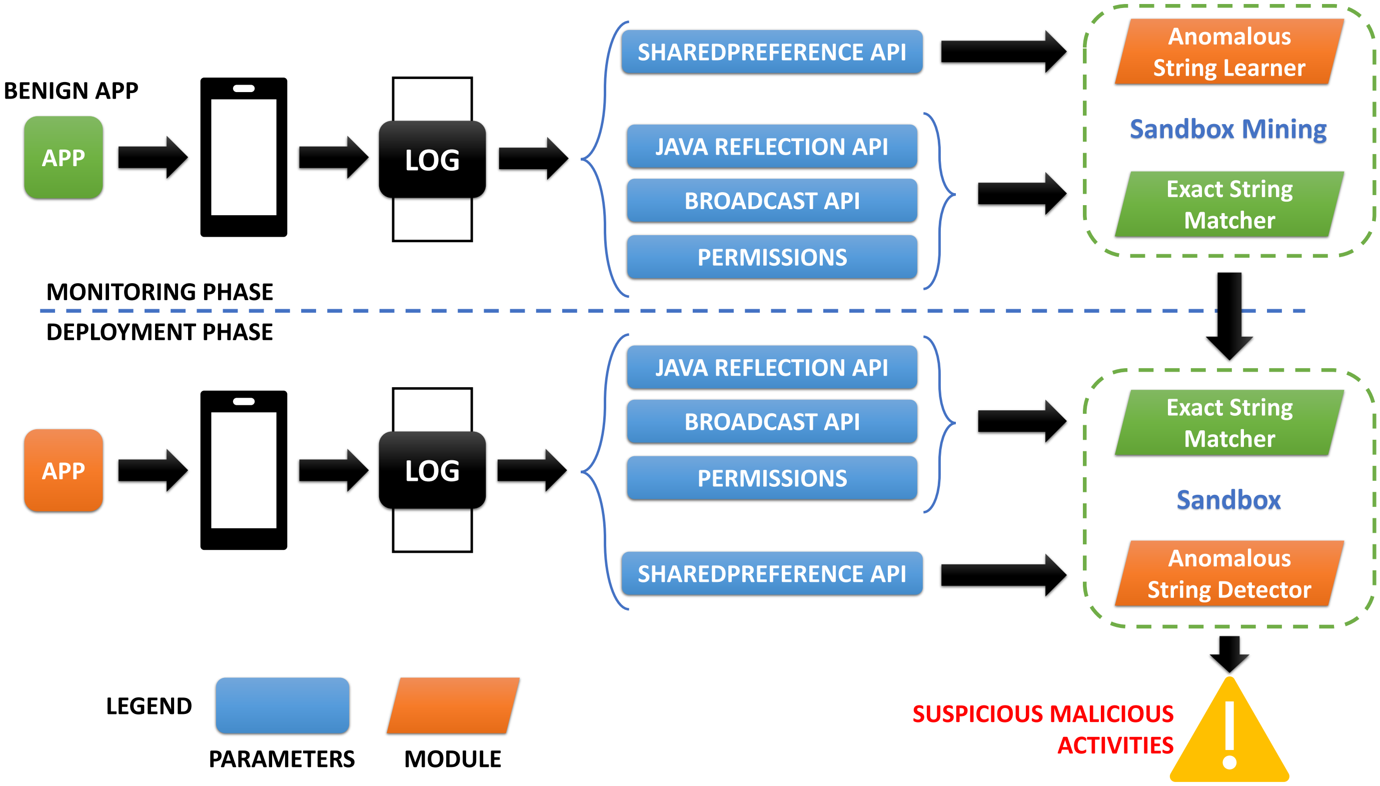

Fig. 1: Overall framework.

whether this pair was seen during the monitoring phase. They found that the *per-event* access control takes longer time to mine sandboxes and increases the risk of false alarms. In our paper, we use these two approaches as baselines. We refer to the per-app strategy and per-event strategy as Boxmate^{A} and Boxmate^{E}, respectively.

C. Automated Test Case Generation Tools for Android

Automated test case generation tools for Android are designed to detect existing faults in apps. There are different strategies to explore the behavior of an app under test. Choudhary et al. [7] highlight three main categories: random (e.g., Monkey [8] and Dynodroid [9]), model-based (e.g., GUIRipper [10], PUMA [11], Droidmate [12], and Droidbot [13]), and systematic (e.g., ACTEve [14] and EvoDroid [15]) strategies. Random tools randomly generate test inputs. On the other hand, model-based tools construct a model from the GUI of an app and explore it to create test cases. Systematic test case generation tools usually utilize complicated and expensive methods (e.g., symbolic execution, evolutionary algorithm) in order to systematically generate test cases that are likely to achieve higher coverage.

Our previous work [16] has investigated the effectiveness of mining sandboxes on detecting malicious behavior using five automated testing tools, i.e., Monkey [8], Droidmate [12], Droidbot [13], GUIRipper [10], and PUMA [11]. We found that Droidbot achieved the best performance on 112 app pairs, so we choose Droidbot as the automatic testing tool in this study.

III. Proposed Approach

A. Overall Framework

Figure 1 describes the overall framework of our sandbox mining approach. Our approach works on two phases: *monitoring* and *deployment*. The monitoring phase analyzes behaviors of a benign version of an Android app to construct a sandbox `SB` that can distinguish benign from anomalous behaviors. The `SB` is then leveraged in the deployment phase to detect anomalous behaviors. If the sandbox detects a potential threat from the app during the deployment phase, it immediately intercepts the operation of the app and raises a warning to an Android user about a potential malicious activity. The following paragraphs describe detailed operations of our approach in monitoring and deployment phases:

1) Monitoring Phase: In this phase, our framework takes as input a benign Android app. We first conduct static analyses on the APK file of the benign app to find Android permissions (e.g., `android.permission.CAMERA`, etc.) that might be required by calls to a number of Android system APIs. In particular, all system calls are extracted and mapped to corresponding permissions[1].

Next, we utilize a GUI test case generation tool, named Droidbot [13], to create a rich set of test cases for discovering

[1] The list of system APIs that require Android permissions is obtained using PScout [17].

TABLE I: Sensitive APIs accepting information of resources as input parameters.

android.webkit.WebView.loadUrl
android.webkit.WebView.loadDataWithBaseURL
java.net.URL.OpenConnection
android.content.ContentResolver.query
android.content.ContentResolver.registerContentObserver

runtime behaviors of the input benign app. Then, our approach executes these test cases on the app, and monitors invocations of specific types of APIs; these API types are SharedPreferences[2], reflection, BroadcastReceiver[3], and sensitive APIs (see Section III-B). These APIs are particularly important since their functionalities can affect security and privacy of app users. During runtime, we record values of specific parameters passed to each monitored API. For example, our approach focuses on parameters containing information of sensitive resources (e.g, file locations, URLs, etc.) input to sensitive APIs (see Table I), key-value string pairs extracted from SharedPreferences objects, etc.

Subsequently, we leverage the extracted parameters and the list of requested permissions to build the core of the resultant sandbox by using two modules named Exact Value Matcher (EVM) and Anomalous Value Detector (AVD). In particular, EVM performs exact string matching, which is suitable for detecting malicious resource accesses or permission requests. On the other hand, AVD (see Section III-C) is built on top of a Support Vector Machine classification model, which is capable of distinguishing malicious from benign latent content hidden inside string parameter values.

The resultant sandbox `SB` is then employed in Deployment Phase to detect anomalous unseen permissions requested from an unknown app or anomalies in parameters passed to vital APIs during runtime.

2) Deployment Phase: This phase employs the constructed sandbox `SB` to detect malicious behaviors in an Android app. EVM and AVD of the sandbox `SB` together monitor requested permissions and parameter values passed to a number of API types during the execution of the Android app. In particular, EVM analyzes permissions and parameters passed to reflection, BroadcastReceiver, and sensitive APIs. On the other hand, AVD processes strings extracted from key-value pairs in SharedPreferences objects. Once an anomalous parameter is detected by EVM or AVD, the sandbox `SB` raises warnings to users regarding the suspicious parameters.

B. Monitored APIs & Extracted Parameters

In both monitoring and deployment phases, our approach monitors invocations of a number of API types during the execution of an Android app. Invocations of these APIs can be exploited by attackers or malware to take control of mobile devices or stole users' data. Monitored APIs and corresponding parameters are classified to the following categories:

Sensitive: These are Android system APIs that can be used to access private or security sensitive data or resources in mobile devices, e.g., device ID, contacts, locations, etc. Malware and attackers can exploit invocations of sensitive APIs to take control of mobile devices. Furthermore, invocations of these APIs may cause private data leakage.

Malicious activities often require specific external resources or data; for example, an adware needs to access a remote server to push an advertisement. Thus, we believe resource information (i.e., file locations or URLs) passed to sensitive APIs provides useful hints to differentiate malicious from benign behaviors. In this work, we use the list of sensitive APIs defined by the AppGuard privacy-control framework [5], which includes a total of the 97 APIs. Among the above sensitive APIs, we manually check and find that 5 out of 97 APIs accept information of resources (e.g., file locations, URLs, etc.) as parameters (see Table I). During execution of an Android app, our approach monitors the above 5 APIs and records their input parameters that contain information of resources (i.e., file locations or URLs) being accessed. We refer to these parameters as *sensitive resource parameters*.

Reflection: In Android programming, reflection APIs belong to `java.util.reflect` package, which can be used to inspect or alter behavior of classes, interfaces, or methods during runtime. For example, these APIs provide utilities to access to private fields of a class or invoke methods dynamically. In fact, many malware can bypass static analysis tools by converting the sequence of method invocations in their malicious injected code to a sequence of reflection API calls [18]. Therefore, our approach focuses on monitoring and recording invocations of methods that are passed as parameters to `java.lang.reflect.Method.invoke`. Intuitively, only invocations of Android system APIs are able to directly access to sensitive resources and users' data. Thus, we further perform a processing step that selects parameters that correspond to Android system APIs (i.e., belonging to `android` package). We refer to these parameters as *reflection parameters*.

BroadcastReceiver: BroadcastReceiver APIs provide utilities for an Android app to register and handle particular system or application events. Each registered BroadcastReceiver object receives `android.content.Intent` based notifications when its corresponding events happen during runtime. For example, an Android app can register for the `HEADSET_PLUG` system event, which is activated when a headset is plugged to the mobile device. Attackers might exploit BroadcastReceiver APIs by creating custom BroadcastReceiver objects for attacking purpose. Therefore, our approach monitors and records concrete types of `android.content.Intent` objects passed as parameters to `android.app.ContextImpl.registerReceiver`. We refer to these parameters as *BroadcastReceiver parameters*.

SharedPreferences: Each SharedPreferences object is linked to a file that stores key-value pairs which can be read or written

[2] `android.content.SharedPreferences`
[3] `android.content.BroadcastReceiver`

by utilities provided by SharedPreferences APIs. These key-value pairs contain specific data or settings of an Android app, and they vary from app to app. Noticeably, data managed by SharedPreferences objects are sharable between different parts of an Android app. Malware or attackers might exploit SharedPreferences APIs to capture sensitive information or inject incorrect data to perform malicious activities or take control of mobile devices. Therefore, our approach monitors and records keys and values that are passed to SharedPreferences APIs, and we refer to these parameters as *SharedPreferences parameters*.

C. Anomalous Value Detector

Anomalous Value Detector (AVD) module is responsible for detecting anomalous strings extracted from key-value pairs in SharedPreferences objects. The construction and utilization of AVD in the monitoring and deployment phases, respectively, are highlighted as follows:

Monitoring Phase: In this phase, AVD analyzes parameters observed in the monitoring phase (see Section III-A). Given a parameter value S, AVD considers all n-grams ($1 \leq n \leq \min\{10, \texttt{len}(S)\}$)[4] of S and their frequencies. For instance, "ab" is a 2-gram of "abab" with frequency of 2. Next, we construct a vector of these n-grams where the value of each dimension corresponds to the occurrence frequency of a specific n-gram. Then, we reduce the dimension of all vectors to L by employing Singular-Value Decomposition (SVD) technique to prevent issues with overfitting and curse of dimensionality [19].

Subsequently, we consider entries of these L-dimensional vectors as features, and input them to One-Class SVM algorithm to learn a classification model that is capable of distinguishing benign from anomalous parameters. In order to achieve the best performance for constructed models, we tune the configurations of One-Class SVM algorithm to optimize the accuracy (i.e., percentage of correct classification) on the training data. In particular, we apply Grid Search to select the best configuration of kernel (i.e., sigmoid, polynomial, rbf, or linear) and many other numeric arguments following leave-one-out cross-validation[5] applied on training data recorded in the monitoring phase. Once the best configuration that achieves the highest accuracy is determined, we apply One-Class SVM with the found configuration on the whole training data. The final constructed model is then used in the deployment phase for anomalous parameter detection.

In our experiments, we leverage the implementation of One-Class Support Vector Machine that is available in scikit-learn[6] (0.19.1). By default, we set $L = 3$.

Deployment Phase: In this phase, AVD processes parameters observed in the deployment phase. Similar to the monitoring phase, AVD converts each parameter S' to a vector where the value of each dimension is the frequency of a n-gram ($1 \leq n \leq \min\{10, \texttt{len}(S')\}$) of the parameter. Next, AVD performs SVD to reduce all vectors' dimensions to L. These L-dimensional vectors are input to the One-Class SVM model constructed in the monitoring phase to predict if a parameter is anomalous compared to the observed benign parameters.

IV. Empirical Evaluation

A. Dataset

To investigate the effectiveness of our approach and the baselines in mining sandboxes, we use benign and malicious app pairs, in which each malicious app in a pair is created by grafting malicious code into its corresponding benign app. In particular, we use a repackaged Android app dataset from Androzoo [3], [4]. The repackaged apps are built by unpacking benign apps and grafting some malicious code to them. Previous studies show that most malware is piggybacked of benign apps, e.g., 80% of the malicious samples in the dataset MalGenome [20] are built through repackaging. The dataset we analyzed contains 15,298 app pairs, and each pair has one benign app and one malicious app, which is repacked on the benign app. All benign and malicious apps are real apps that have been released to various app markets.

Unfortunately, not all apps in the dataset can be used in our study. First, we fail to install a number of them on the emulator used in our study due to various compatibility issues. These incompatibilities cause errors to be thrown or apps end gracefully right after they are started. There are 7,490 app pairs left after removing the incompatible apps.

As we want to include different types of malwares in our study, we first determine malware types (e.g., adware, trojan, worm, etc.) of the malicious apps in the remaining app pairs by using Euphony [21], a tool that can infer malware labels for malicious apps. We exclude 1,412 pairs of which the malicious app has empty malware type as inferred by Euphony. Out of the remaining 6,078 app pairs, 5,450 malicious apps are repackaged with adware. Note that a benign app can be repacked with multiple kinds of malwares at the same time. Since it is impossible to run all the pairs of apps in limited time and resources, we randomly choose 25 pairs of apps. Ten of them are repackaged with adware and 15 out of them are repackaged with other malware types.

Table II presents the detail information of these selected 25 pairs of apps. The columns in the table correspond to short acronyms that we use to refer to the app pairs (Pair Index), package names of the app pairs (Package), descriptions of the functionalities of the app pairs (Functionality) and the malware category (Malware). Out of these app pairs, two pairs have the same benign app, i.e., P12 and P13. In this table, the benign apps in the first ten pairs are all repackaged with adware, while there are multiple categories of malware in the last fifteen pairs, including trojan, spyware, riskware, worm, etc. These 25 pairs of apps have different functionalities. For example, P2, P8, P10, P17 and P22 are game apps; P4, P11 and P16 can play audio; P12, P13, and P15 can display pictures.

[4] $\texttt{len}(S)$ is the number of characters in S.

[5] We repeatedly select one data instance as test data, and leverage the other ones as training data.

[6] http://scikit-learn.org/stable/index.html

TABLE II: Twenty five pairs of malicious-benign apps used in our study.

Pair Index	Package	Functionality	Malware
P1	com.android.remotecontrolppt	Office remote control	adware
P2	com.dseffects.VirtualDog	A virtual dog game	adware
P3	com.northpark.beautycamera	Take selfie photos	adware
P4	ix.com.android.VirtualRecorder	Record voice and play	adware
P5	mobi.infolife.cachepro	Clean cache in devices	adware
P6	oms.sdt	Flashlight with multiple color	adware
P7	com.argentina.argentina	A mail app in Argentina	adware & addisplay
P8	com.androgames.BubbleBurst	A bubble game	adware & trojan
P9	com.omesoft.nosick	A medical app for motion sickness	adware & addisplay & trojan & riskware
P10	com.vinanetjsc.dkamphuocFilix	A zombie game	adware & addisplay & trojan & spyware
P11	byappy.icallmeon	Remind users with text or voice messages	riskware & spyware
P12, P13	com.AirplanesWallpapers.BoeingE3ASentryWallpaper	Wall paper gallery	spyware
P14	com.applacarte.a13701917511300513987	Optimist investment service	trojan
P15	com.droidappik.sexyhotxxxgirls	Sexy girls bikini picture gallery	trojan
P16	com.hedami.musicplayerremix	Play music	worm
P17	com.onepixelarmy.fastball	A casual arcade game	trojan
P18	com.yy.fontmaster	Manage different fonts	exploit & trojan
P19	net.halfmobile.scannerfree	A handy scanner for PDF creator	riskware
P20	org.apache.cordova.rotarybook	Books & reference	trojan & virus
P21	cz.kinst.jakub.clockq	A simple digital clock widget	trojan
P22	com.pommedeterresautee.angrybirdsseasonunlock	AngryBirds Seasons unlock	trojan
P23	appinventor.ai_rintoadi.RadioICBB	Radio Islamic Centre Bin Baz	trojan & spyware
P24	lysesoft.andftp	A FTP client for Android	trojan & fakelook & unclassifiedmalware
P25	com.nextminute.app	Job management	trojan & dropper & virus

B. Experimental Settings

1) Automated Test Case Generation Tools: In our study, we choose Droidbot as the automated test case generation tool to run the selected apps. Recently, Bao et al. [16] find that Droidbot is more effective in constructing sandboxes than four other automated test case generation tools, i.e., Monkey, Droidmate, GUIRipper, and PUMA. Moreover, both Choudhary et al. [7] and Bao et al. [16] show that most of automated test case generation tools achieve similar code coverage. Thus, we believe Droidbot is a good choice in our study. In our experiments, we execute Droidbot three times with different initial seeds in order to generate three different sets of test cases. Our goal is to ensure all execution scenarios are covered by Droidbot as well as to facilitate the computation of False Alarm Rate (see Section IV-B3).

Droidbot[7] is an open-source model-based test case generation tool that provides utilities for easy installation. We use the same setting as Bao et al. [16]'s study. In particular, we configure the emulator to run Android SDK version 19. Our emulator is a version of Genymotion emulator[8] with 2GB of RAM and Intel x86 CPU architecture. The emulator is executed on a Mac OS 10.12 (Sierra) laptop running Intel Core i5 CPU (2.3 GHz).

2) Instrumentation and Trace Collection: To record parameter values passed to interested APIs during the execution of Android apps on generated test cases, we employ Droidmon[9], which is an open-source Dalvik monitoring framework. Different from the other Android instrumentation tools (e.g., DroidFax [22], etc.), Droidmon requires no modifications in the bytecode of the APK files for injecting additional instrumentation code. Furthermore, Droidmon is based on Xposed Framework[10], which can change behaviors of Android system and apps without touching APK files. Hence, Droidmon provides the utilities to operate on different Android versions without requiring additional efforts.

Droidmon requires a predefined input list of APIs in order to record their invocations during runtime of Android apps. We select and input the list of APIs highlighted in Section III-B to Droidmon as parameters passed to these APIs contain important information for building sandboxes. Every time an interested API is invoked, Droidmon leverages `logcat`[11] to store the invocation of that API. Droidmon eventually produces a JSON file containing information of invoked APIs of interest and their parameters.

3) Evaluation Metrics: We assess the effectiveness of a sandbox in detecting malicious apps by leveraging the following metrics:

- **True Positives (TP):** Number of malicious apps that are classified as malicious.
- **True Negatives (TN):** Number of benign apps that are classified as benign.
- **False Negatives (FN):** Number of malicious apps that are classified as benign.
- **False Positives (FP):** Number of benign apps that are classified as malicious.

Then, we use the values of True Positives and False Negatives to compute the **True Alarm Rate (TAR)**, which is the

[7] https://github.com/honeynet/droidbot

[8] https://www.genymotion.com/

[9] https://github.com/idanr1986/droidmon

[10] http://repo.xposed.info/

[11] `logcat` is an official tool that dumps a log of system messages as well as messages from user apps with Android's Log class.

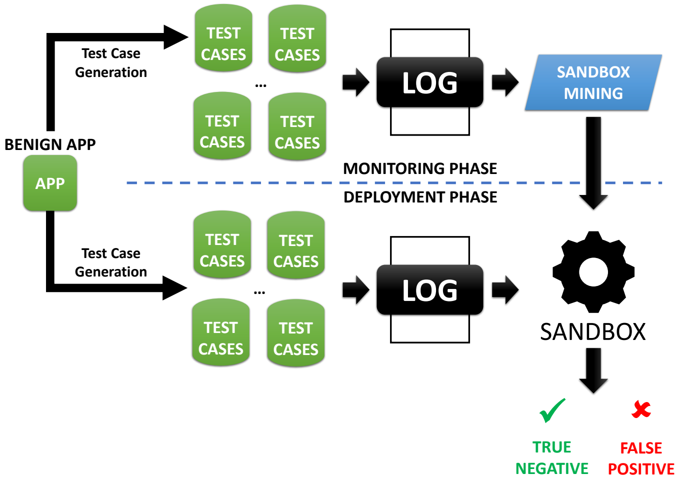

Fig. 2: Framework to determine False Alarm Rate.

percentage of apps that are correctly detected as malicious:

$$\text{TAR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1}$$

The higher the values of TAR, more of the malicious apps are detected as such. Next, we leverage recorded values of True Negatives and False Positives to calculate **False Alarm Rate (FAR)**, which is the percentage of apps that are incorrectly detected as malicious:

$$\text{FAR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{2}$$

The smaller the values of FAR, the smaller the probability that false alarms are generated. We also compute **Precision**, **Recall**, and **F-measure** to evaluate effectiveness of mined sandboxes in classifying an app as *benign* or *malicious* as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F-measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

All of the above metrics are popular and widely adopted by state-of-the-art approaches that detect malwares and privacy leaks in Android apps (e.g., [23]–[27]).

In order to compute True Positives (TP) and False Negatives (FN) of a mining sandbox approach, for each pair of benign and malicious app (see Table II), we construct a sandbox from recorded data of the benign app in monitoring phase, and deploy the constructed sandbox on its corresponding malicious app. Following Equation 1, we use True Positives (TP) and False Negatives (FN) to compute True Alarm Rate (TAR). On the other hand, to estimate True Negatives (TN) and False Positives (FP), for each benign Android app we perform cross-validation among the three different sets of test cases generated by three distinct execution of Droidbot (see Figure 2). In particular, we use monitored data of two sets of test cases as training data to construct a sandbox. Then, we deploy the inferred sandbox on recorded data of the remaining set of test cases to check whether the resultant sandbox can detect benign apps as malicious (i.e., false positive). We repeat this process three times each using a different set of test cases to detect false positives. In total, we analyze 3 x 25 = 75 combinations between benign apps and the sets of generated test cases. Then, we employ values of True Negatives (TN) and False Positives (FN) to compute False Alarm Rate (FAR) (see Equation 2).

4) Baselines: We compare our proposed approach with two variants of Boxmate [2], named Boxmate$^{\text{A}}$ and Boxmate$^{\text{E}}$. In particular, Boxmate$^{\text{A}}$ considers execution of sensitive APIs in benign apps to construct sandboxes. On the other hand, Boxmate$^{\text{E}}$ additionally takes into account GUI elements that call sensitive APIs when inferring sandboxes. During its operation, Boxmate$^{\text{E}}$ is capable of recording executed sensitive APIs after an event trigger, i.e., an interaction with GUI elements such as touching a text label, clicking a button, etc. Subsequently, Boxmate$^{\text{E}}$ constructs a sandbox based on the pairs of events and sensitive APIs. This per-event access control is much more fined-grained, which increases the risk of false alarms. For a fair comparison, the same set of test cases and evaluation methodology are considered for our proposed approach and the baselines.

C. Research Questions

Research Question 1: How effective is our proposed sandbox mining approach compared to the baselines?

In this research question, we investigate the effectiveness of our proposed approach in terms of different evaluation metrics (see Section IV-B3). Then, we compare our approach with two baselines, named Boxmate$^{\text{A}}$ and Boxmate$^{\text{E}}$ (see Section IV-B4).

Research Question 2: Which type of recorded data contributes most to the effectiveness of inferred sandboxes in detecting malicious behaviors?

In this research question, we inspect the impact of each type of recorded data (i.e., sensitive resource, reflection, BroadcastReceiver, SharedPreference parameters, and Android permissions) on the effectiveness of constructed sandboxes. In particular, we exclude each type of recorded data and construct a number of new sandboxes. Then, we observe the effectiveness of these sandboxes in detecting malicious behaviors compared to the default setting.

Research Question 3: How effective is Exact Value Matcher (EVM) compared to Anomalous Value Detector (AVD)?

In this research question, we investigate the effectiveness of EVM and AVD when they operate independently. We first set each of the two modules to process all recorded data (i.e., sensitive APIs, reflection, BroadcastReceiver, SharedPreference parameters, and Android permissions) to detect anomalous activities. Then, we compare the effectiveness of EVM and AVD with the original setting.

D. Findings

a) RQ1 – Effectiveness of Proposed Approach: Table III shows the effectiveness of our approach and the two baselines

TABLE III: Effectiveness of our approach and the two baselines. "FR" represents False Alarm Rate, and "TR" stands for True Alarm Rate; "P" is Precision, "R" is Recall, and "F" is F-measure; "BM$^{\text{A}}$" is Boxmate$^{\text{A}}$, "BM$^{\text{E}}$" is Boxmate$^{\text{E}}$, and "OA" stands for our approach.

Approach	TR(%)	FR(%)	P(%)	R(%)	F(%)
BM$^{\text{A}}$	16.00	9.33	36.36	16.00	22.22
BM$^{\text{E}}$	80.00	58.67	31.25	80.00	44.94
OA	80.00	5.33	83.33	80.00	81.63

(i.e., Boxmate$^{\text{A}}$ and Boxmate$^{\text{E}}$) on 25 pairs of Android apps considering various evaluation metrics. From the table, we note that our approach achieves the highest True Alarm Rate (i.e., 80%), lowest False Alarm Rate (i.e., 5.33%), as well as highest Precision (i.e., 83.33%), Recall (i.e., 80.00%), and F-measure (i.e., 81.63%). Noticeably, our approach outperforms the best baselines by 42.87%, 129.18%, and 81.64% in terms of False Alarm Rate, Precision, and F-measure, respectively. Overall, our approach achieves the best performance compared to the two variants of Boxmate.

b) RQ2 – Contributions of Extracted Parameters & Permissions: Table IV shows the effectiveness of our proposed approach considering various subsets of extracted parameters and permissions. From the table, we find that the exclusion of sensitive resource parameters significantly reduce the effectiveness of our approach in terms of all evaluation metrics compared to the default setting. Furthermore, we note that extracted permissions have substantial impact on the effectiveness on our approach since TAR, Precision, Recall, and F-measure decrease when permissions are excluded. Overall, sensitive resource parameters and permissions have a high impact on the effectiveness of our sandbox mining approach.

We find that exclusion of parameters passed to reflection and BroadcastReceiver APIs has no significant impact on the effectiveness our approach. We still retain these parameters since our dataset of Android app pairs might not be large enough to capture the importance of these two types of parameters. Moreover, previous research studies show that Android malware can use reflection and BroadcastReceiver APIs to perform harmful activities [28], [29]. For example, Aafer et al. [28] highlight that Android malware may utilize reflection APIs to easily obfuscate dangerous API calls and thus bypass static analyses employed to identify these dangerous API invocations. Additionally, Feng et al. [29] show that a broadcast receiver can be used by malwares to launch a service upon the completion of system events (e.g., incoming SMS messages or outgoing calls) for forwarding users' private information (e.g., IMEI number, subscriber id, etc.) to a remote server.

c) RQ3 – AVD vs. EVM: Table V shows the effectiveness of Exact Value Matcher (EVM) and Anomalous Value Detector (AVD). From the table, we note that EVM results in a higher False Alarm Rate than the default setting and AVD (by 50.09%). On the hand, AVD has lower True Alarm Rate than EVM and the default setting (by 20%). Overall, we believe that EVM and AVD achieve the best performance when they are combined together.

TABLE IV: Contributions of extracted parameters and permissions. "FR" represents False Alarm Rate, and "TR" stands for True Alarm Rate; "P" is Precision, "R" is Recall, "F" is F-measure, and "OA" stands for our approach.

Approach	TR(%)	FR(%)	P(%)	R(%)	F(%)
OA − Resource	48.00	4.00	80.00	48.00	60.00
OA − Reflection	80.00	5.33	83.33	80.00	81.63
OA − BroadcastReceiver	80.00	5.33	83.33	80.00	81.63
OA − SharedPreference	76.00	5.33	82.61	76.00	79.17
OA − Permission	72.00	5.33	81.82	72.00	76.60
OA	80.00	5.33	83.33	80.00	81.63

TABLE V: Effectiveness of Exact Value Matcher (EVM) and Anomalous Value Detector (AVD). "FR" represents False Alarm Rate, and "TR" stands for True Alarm Rate; "P" is Precision, "R" is Recall, and "F" is F-measure.

Approach	TR(%)	FR(%)	P(%)	R(%)	F(%)
EVM	80.00	8.00	76.92	80.00	78.43
AVD	64.00	5.33	80.00	64.00	71.11
EVM + AVD	80.00	5.33	83.33	80.00	81.63

E. Threats to Validity

Internal Validity. Threats to internal validity relate to errors in implementation. We carefully inspected our scripts and source code, but there might be errors that we missed. There are also potential threats related to accuracy of Android permission mappings to system APIs. In our experiments, we used mappings[12] provided by PScout [17]. We have manually checked these mappings, but there could be wrong or missed mappings that we did not notice.

External Validity. Threats to external validity correspond to the generalizability of our findings. Our study only leverages Droidbot to generate test inputs for mining sandboxes. For each app, we execute Droidbot with time limit of one hour to discover runtime behaviors of the app. Additionally, we only analyze 25 pairs of benign and malicious apps due to limited resources. Still, our number of investigated apps is comparable to those examined by past studies that also conduct dynamic analysis on Android apps [10], [14], [30]. All malicious apps in our work are collected from the piggybacked app dataset released by Li et al. [31]. These piggybacked apps are not guaranteed to cover all types of Android malware. Still, most malwares are piggybacked of benign apps; for instances, 80% of malicious samples in MalGenome [20] are created by repackaging.

As future work, we plan to include more Android apps and malwares as well as leverage more automated test case generation tools that are popular in industry and academia in order to reduce these threats to external validity.

Construct Validity. Threats to construct validity relate to our evaluation metrics. In our work, we have followed state-of-the-art approaches in malware and privacy leak detection

[12] http://pscout.csl.toronto.edu/

(e.g., [23]–[27]) that also use False Alarm Rate, True Alarm Rate, Precision, Recall, and F-measure as evaluation metrics.

V. Related Work

A. Sandboxing

The mining sandboxes on Android named Boxmate is first proposed by Jamrozik et al. [2]. In their work, they evaluated the effectiveness of Boxmate and demonstrated its low false alarm rate on several benign apps downloaded from Google Play. They found that the set of sensitive APIs were quickly saturated by automated test generation. They also found that there were few false alarms by checking Boxmate against 18 use cases reflecting typical app usage.

Jamrozik et al. did not validate the effectiveness of sandbox mining with real malware. In our previous study [16], we conducted an empirical study using five automated test case generation tools to demonstrate the effectiveness of mining sandboxes on detecting malicious apps. In that work, we found that the sandboxes constructed based on sensitive APIs called from benign apps can effectively detect more than 75% malicious apps in the selected app pairs. Additionally, among these automated test case generation tools, Droidbot can detect the most malicious apps. However, that work did not consider parameter values in the called APIs. In this work, we extend our previous work by building more effective sandboxes by combining multiple categories of APIs with string parameters.

There are several other works on developing and analyzing sandboxes. For example, Cappos et al. [32] proposed a more secure sandbox with a security layer that can prevent attackers from leveraging bugs in privileged functionalities. Also, Graziano et al. [33] proposed a technique to analyze sandboxes that were available as public online services to identify malware development activities in those sandboxes so that preventive actions can be taken early. Wan et al. [34] also use automated testing to extract system API calls of operating systems and build a sandbox for Linux containers. Different from our work, these studies either do not consider mobile apps or do not construct sandboxes by analyzing app behaviors.

B. Automated Test Case Generation for Android

Many automated test case generation tools have been proposed for Android apps. Choudhary et al. [7] have performed a comparative study to evaluate six automated test case generation tools, i.e., Monkey [8], ACTEve [14], Dynodroid [9], A3E-Depth-first [35], GUIRipper [10], and PUMA [11]. They also divided these automated test case generation tools into three major behavior exploration strategies, i.e., random exploration [8], [9], [11], [12], model based exploration [10], [13], [30], [35]–[38], and systematic exploration [15], [39], [40].

Monkey [8] is a typical testing tool that uses the random exploration strategy. It can generate pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. As a part of Android developer toolkit, it is easy to use and highly compatible with different Android versions. Machiry et al. [9] proposed Dynodroid, which leverages a novel "observe-select-execute" strategy to efficiently generate random events and select the ones related to current execution states of the apps.

There are a number of automated testing tools using a model based exploration strategy. Amalfitano et al. [10] developed GUIRipper that systematically explores GUIs of apps by maintaining state-machine models of GUIs, named GUI Tree models. Droidmate proposed by Jamrozik et al. implements a GUI-state based exploration strategy, which is inspired by Dynodroid [9]. The key idea of its exploration strategy is to interact with views (GUI elements) randomly, but give precedence to views that have been interacted the least amount of times so far. Droidbot [13] dynamically builds a GUI model of an app under test by collecting GUI information and running process information. Different from many other model-based tools, Droidbot is lightweight and does not require system modification or app instrumentation. Our previous study has shown that Droidbot can identify more malicious apps than four other automated test case generation tools [16].

Systematic exploration strategy usually relies on some sophisticated techniques (e.g., symbolic execution and evolutionary algorithms) to reveal some application behavior upon specific inputs. Anand et al. [14] presented a new technique, named ACTEve, that employs concolic execution for generating sequences of events for Android applications with available source code. Mahmood et al. [15] introduced an evolutionary algorithm based testing framework, named EvoDroid, for generating relevant test cases for Android application.

VI. Conclusion and Future Work

In this work, we propose a sandbox mining approach that analyzes parameters passed to different types of APIs as well as requested Android permissions to construct more accurate sandboxes for malware detection. We leverage an automated test case generation tool, named Droidbot [13], to generate a rich set of GUI test cases for exploring behaviors of benign apps. During execution of these test cases, we analyze parameters passed to four different types of APIs: *sensitive*, *reflections*, *BroadcastReceiver*, and *SharedPreferences* APIs. Furthermore, we extract the set of Android permissions. We utilize the collected parameters and requested Android permissions to construct a sandbox that can identify anomalous activities which are potentially harmful to Android users. We conduct our experiments on 25 pairs of malicious and benign apps extracted from piggybacked apps dataset released by Li et al. [31]. The empirical results indicate that our approach is more effective than the best performing variant of Boxmate [2] by 81.64% in terms of F-measure.

As future work, we plan to improve the effectiveness of our approach further by considering additional types of parameters and APIs aside from the ones used in this work. We also plan to leverage many different automated test cases generation tools to create a richer set of test cases to better construct sandboxes.

Acknowledgment

This research was supported by the Singapore National Research Foundation's National Cybersecurity Research & Development Programme (award number: NRF2016NCR-NCR001-008).

References

[1] H. T. T. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, and S. Bhattacharya, "The company you keep: mobile malware infection rates and inexpensive risk indicators," in *Proc. of WWW*, 2014.

[2] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proc. of ICSE*, 2016, pp. 37–48.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proc. of MSR*. IEEE, 2016, pp. 468–471.

[4] L. Li, J. Gao, M. Hurier, P. Kong, T. F. Bissyandé, A. Bartel, J. Klein, and Y. L. Traon, "Androzoo++: Collecting millions of android apps and their metadata for the research community," *CoRR*, vol. abs/1709.05281, 2017. [Online]. Available: http://arxiv.org/abs/1709.05281

[5] 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*. Springer, 2014, pp. 213–231.

[6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner, "Android permissions demystified," in *Proc. of CCS*, 2011, pp. 627–638.

[7] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *Proc. of ASE*. IEEE, 2015, pp. 429–440.

[8] "https://developer.android.com/studio/test/monkey.html," accessed: October 2017.

[9] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," in *Proc. of FSE/ESEC*, 2013.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *Proc. of ASE*, 2012.

[11] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.

[12] 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)*, 2016, pp. 293–294.

[13] 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 Companion*. IEEE Press, 2017.

[14] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. of FSE*, 2012, p. 59.

[15] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proc. of FSE*. ACM, 2014, pp. 599–609.

[16] L. Bao, T.-D. B. Le, and D. Lo, "Mining sandboxes: Are we there yet?" in *Proc. of SANER*. IEEE, 2018, pp. 445–455.

[17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. of CCS*, 2012, pp. 217–228.

[18] 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*, 2016.

[19] E. Keogh and A. Mueen, "Curse of dimensionality," in *Encyclopedia of machine learning*. Springer, 2011, pp. 257–258.

[20] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[21] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware," in *Proc. of MSR*. IEEE, 2017, pp. 425–435.

[22] H. Cai and B. G. Ryder, "Droidfax: A toolkit for systematic characterization of android applications," in *Proc. of ICSME*. IEEE, 2017.

[23] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[24] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *Proc. of NDSS*, 2014.

[25] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proc. of CCS*, 2014.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. of PLDI*, 2014.

[27] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proc. of ICSE*, 2015.

[28] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks*, 2013.

[29] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: semantics-based detection of android malware through static analysis," in *Proc. of FSE*, 2014.

[30] W. Choi, G. C. Necula, and K. Sen, "Guided GUI testing of android apps with minimal restart and approximate learning," in *Proc. of OOPSLA*, 2013.

[31] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[32] M. Graziano, D. Canali, L. Bilge, A. Lanzi, and D. Balzarotti, "Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence," in *Proc. of USENIX*, 2015.

[33] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. E. Anderson, "Retaining sandbox containment despite bugs in privileged memory-safe code," in *Proc. of CCS*, 2010, pp. 212–223.

[34] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *Proc. of ICST*. IEEE, 2017, pp. 92–102.

[35] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[36] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[37] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE) 2013*, 2013.

[38] Y. M. Baek and D. Bae, "Automated model-based android GUI testing using multi-level GUI comparison criteria," in *Proc. of ASE*, 2016.

[39] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proc. of ISSTA*, 2013.

[40] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proc. of NDSS*, 2016.