Mining Sandboxes: Are We There Yet?

Lingfeng Bao, Tien-Duy B. Le, David Lo
School of Information Systems
Singapore Management University, Singapore
{lfbao, btdle.2012, davidlo}@smu.edu.sg

***Abstract*—The popularity of Android platform on mobile devices has attracted much attention from many developers and researchers, as well as malware writers. Recently, Jamrozik *et al.* proposed a technique to secure Android applications referred to as *mining sandboxes*. They used an automated test case generation technique to explore the behavior of the app under test and then extracted a set of sensitive APIs that were called. Based on the extracted sensitive APIs, they built a sandbox that can block access to APIs not used during testing. However, they only evaluated the proposed technique with benign apps but not investigated whether it was effective in detecting malicious behavior of malware that infects benign apps. Furthermore, they only investigated one test case generation tool (i.e., Droidmate) to build the sandbox, while many others have been proposed in the literature.**

In this work, we complement Jamrozik et al.'s work in two ways: (1) we evaluate the effectiveness of mining sandboxes on detecting malicious behaviors; (2) we investigate the effectiveness of multiple automated test case generation tools to mine sandboxes. To investigate effectiveness of mining sandboxes in detecting malicious behaviors, we make use of pairs of malware and benign app it infects. We build a sandbox based on sensitive APIs called by the benign app and check if it can identify malicious behaviors in the corresponding malware. To generate inputs to apps, we investigate five popular test case generation tools: Monkey, Droidmate, Droidbot, GUIRipper, and PUMA. We conduct two experiments to evaluate the effectiveness and efficiency of these test case generation tools on detecting malicious behavior. In the first experiment, we select 10 apps and allow test case generation tools to run for one hour; while in the second experiment, we select 102 pairs of apps and allow the test case generation tools to run for one minute. Our experiments highlight that 75.5%–77.2% of malware in our dataset can be uncovered by mining sandboxes – showing its power to protect Android apps. We also find that Droidbot performs best in generating test cases for mining sandboxes, and its effectiveness can be further boosted when coupled with other test case generation tools.

***Index Terms*—Mining Sandboxing, Android Malware, Automated Test Case Generation**

I. Introduction

Android has become the most dominant mobile platform today. A report from Gartner highlighted that 81.7% of mobile devices run on Android platform in the 4^{th} quarter of 2016[1]. With the availability of a huge number of Android apps in multiple marketplaces (e.g., Google Play), users are given a wide range of options to select useful apps for their work and entertainment. Unfortunately, mobile devices running on Android are increasingly targeted by attackers. Truong *et al.* reported that around 0.25% of Android devices were infected with malware [1], which is still a large number in consideration of the total number of Android devices. To secure Android app users, Android platform provides a permission based mechanism to protect sensitive resources (e.g., contacts, networks, locations, etc.). However, these permissions are often too *coarse-grained* to prevent malicious behaviors.

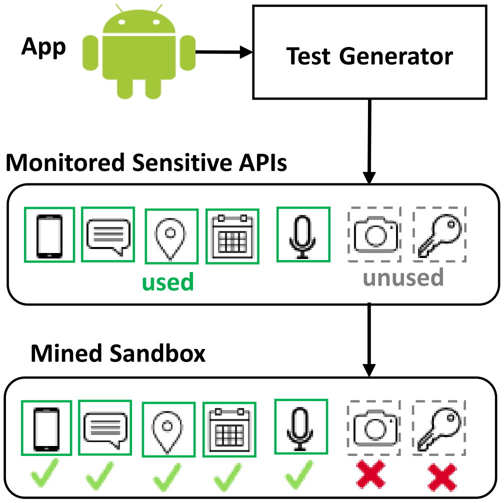

Fig. 1. Mining Sandbox.

Recently, Jamrozik *et al.* proposed Boxmate [2] to secure Android apps by *mining sandboxes* that can protect resources at a *fine-grained* level by limiting access to sensitive APIs. Boxmate can protect against unexpected behaviors introduced by either an attacker "injecting malware" to a benign app[2] or introducing a malicious app that looks benign into an app market. Figure 1 presents the process of Boxmate for mining sandboxes. This technique first employs Droidmate [4], which is an automated test case generation tool, to explore behaviors of an Android app. During its operation, Boxmate identifies sensitive Android APIs called during the execution of test cases and uses them to form a sandbox. The sandbox will then block calls to sensitive APIs unseen during testing. They evaluated the proposed technique using twelve apps from the top downloads of the Google Play Store. They found that the set of sensitive APIs were quickly saturated by automated test

[1] https://www.gartner.com/newsroom/id/3609817

[2] Xiao et al. have demonstrated that "an attacker can easily insert malicious codes into any valid APK, without breaking its signature" using various attack mechanisms [3].

generation. They also found that there were few false alarms by checking Boxmate against 18 use cases reflecting typical app usage.[3]

Jamrozik et al.'s work still leave some room for future work. First, Jamrozik et al. only evaluated the proposed technique using twelve benign apps. There was no malware evaluated in their experiment. Thus, there is a need for additional studies to further demonstrate the applicability of mined sandboxes to detect and prevent malicious behaviors. Second, Jamrozik et al. only used one automated test case generation tool, i.e., Droidmate [4], to build a sandbox. Since there are many automated test case generation tools proposed in the literature, e.g., [5], [6], [7], [8], [9], [10], [11], [12], [13], more investigation is needed to assess: (1) which among these test case generation tools is more effective for building sandboxes, and (2) whether they can be used together to mine better sandboxes.

This study aims to address the above mentioned needs. In particular, we investigate (1) effectiveness of mining sandboxes in detecting malicious behaviors, and (2) effectiveness of multiple automated test case generation tools to mine sandboxes. Our study makes use of pairs of malware and benign app it infects. We build a sandbox based on sensitive APIs called by the benign app and check if it can identify malicious behaviors in the corresponding malware. To generate inputs to apps, we investigate five popular and state-of-the-art test case generation tools: one tool from industry (i.e., Monkey [14]) and four tools from academia (i.e., Droidmate [4], Droidbot [15], GuiRipper [7], PUMA [10]).

We conduct two experiments in this work. One of the experiments is computationally expensive and only considers a small number of app pairs (`SmallE`), while another is computationally inexpensive and considers a larger number of app pairs (`LargeI`). In the first experiment, we run the selected test case generation tools on 10 app pairs for one hour. Each pair contains one malicious app and one benign app it infects. We instrument all the tested apps using a tool named DROIDFAX [16] for collecting the API traces. In the second experiment, we run these selected tools on 102 pairs of apps for only one minute. All the apps used in our study are from a real life piggybacked Android app dataset collected by Li *et al.* [17]. Piggybacked apps are built by attackers by unpacking benign apps and then grafting some malicious code to them. Most malware is piggybacked of benign apps, e.g., 80% of the malicious samples in the MalGenome dataset [18] are built through repackaging.

The first experiment shows that 8 out of 10 malicious apps can be detected by the sandbox constructed by combining all the automated test case generation tools, which indicates the power of mining sandboxes in protecting apps. We notice that there is only little variation in the effectiveness of the test case generation tools; the numbers of malicious apps detected by sandboxes constructed by running Monkey, Droidmate, Droidbot, GUIRipper, and PUMA were 7, 6, 6, 6, and 5, respectively. We also find that all these tools except Monkey can detect the malicious behavior within a short amount of time (i.e., less than one minute). This indicates that sandboxes built by the test case generation tools can detect malicious apps efficiently. Then, in the second experiment, we find that 75.5% (77 out of 102) of malicious apps are detected by the sandbox constructed by combining all these tools. Among these tools, the sandbox constructed by running Droidbot had the best performance, i.e., 68 malicious apps were detected. We also find that if we combined Droidbot with another tool, the number of detected malicious apps was highly increased, i.e., 73, 74, 77, and 71 for Monkey, Droidmate, GUIRipper, and PUMA, respectively.

This paper makes the following main contributions:

- We evaluate mining sandboxes with malware that infects benign apps. We conduct two experiments with a considerable amount of malware and benign apps they infect, i.e., 10 app pairs and 102 app pairs, respectively.
- We investigate the effectiveness of five test case generation tools to construct sandboxes. Our experiments highlight that the sandboxes constructed by running these tools can detect malicious apps effectively, i.e., 8 out of 10 malicious apps in the first experiment and 77 of the 102 malicious apps in the second experiment are successfully detected. Also, composition of multiple test case generation tools can boost the effectiveness of constructed sandboxes.

The remainder of the paper is structured as follows. Section II presents background materials on Android, sandboxing, and five automated test case generation techniques. Section III describes the experiment setup. Section IV presents the experiment results. Section V discusses some implications and threats to validity of this work. Section VI reviews related work. Section VII concludes the paper and discusses future directions.

II. BACKGROUND

A. Android

Android applications are mainly written in Java then compiled into Java bytecode and finally converted into Dalvik bytecode in *dex* file format. The *dex* file, native code (if any), and other resource files are packaged into an *APK* file for distribution and installation. Despite being GUI-based and mainly written in Java, Android apps significantly differ from Java standalone GUI applications. Android apps have no main methods but many entry points that are methods implicitly called by the Android framework. The Android OS defines a complete lifecycle for all components in an app. There are four different kinds of components an app developer can define: *Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*; these are the top-level abstractions of user interface, background service, response to broadcasts, and data storage, respectively. The Android framework communicates with applications and manages application executions via various

[3] Jamrozik et al. also evaluated the effectiveness of a stricter variant of their proposed approach that involves per-event access control (wrt. the 18 use cases), and the readability of the mined sandbox (through a qualitative study of a sandbox mined from Snapchat). In this work, we focus on per-app access control and do not consider readability of mined sandboxes.

callbacks, including lifecycle methods and event handlers. The inter-component communication (ICC) in Android OS is via passing messages called *intents*. ICCs could be explicit (i.e., the targeted component is specified in the intent) or implicit (i.e., determined by the Android framework at runtime).

There are some resources or data that are deemed private or security sensitive in mobile devices, e.g., device ID, contacts, locations, etc. Android provides a permission mechanism to protect these sensitive data; that is, an app is only allowed to call certain APIs accessing a particular sensitive data (or resource), if it has obtained an explicit permission governing access to the sensitive data, from an authorized user of the Android device where the app is run on. These sensitive APIs often include operations that are security-critical as they may lead to private data leakage. In our study, we use the set of sensitive APIs defined in the AppGuard privacy-control framework [19]; it declares a total of 97 APIs that allow access to crucial private data (or resources) that an average user should be concerned about as sensitive.

B. Sandboxing

A sandbox is an environment in which the actions of a guest application are restricted according to a security policy. Typically, it provides a tightly controlled set of resources (e.g., disk, memory, network access, etc.) for the guest applications to run in. 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 would actually require – this causes the issue of *overprivileged* apps. Felt *et al.* reported that 33% of Android apps were *overprivileged* [20]. One reason that causes overprivileged apps is the fact that the official Android documentation for APIs and permissions is incomplete [21]. To address this limitation, recently, Jamrozik *et al.* proposed novel approach, referred to as *mining sandboxes*; the proposed approach: (1) runs automated test case generation tools to generate test cases that are used to explore a target app; (2) monitors sensitive APIs that are called during the execution of the test cases; (3) uses the set of sensitive APIs as a sandbox that can be deployed to prevent execution of additional sensitive APIs. Thus, the constructed sandboxes of Jamrozik et al. are capable of detecting and preventing unexpected changes in app behaviors. In our study, we want to investigate whether the proposed sandboxing technique is effective to detect malicious behavior in malware, as well as investigate the effectiveness of several automated test case generation tools in constructing sandboxes.

C. Automated Test Case Generation Tools for Android

There are a number of automated test generation tools for Android proposed in the literature. The primary goal of these tools is to detect existing faults in Android apps. The test case generation tools could have different strategies to explore the behavior of an app under test. Choudhary *et al.* [22] have performed a comparative study to evaluate six automated test case generation tools, i.e., Monkey [14], ACTEve [23], Dynodroid [5], A^{3}E-Depth-first [8], GUIRipper [7], and PUMA [10]. They put these tools into three categories, i.e., random, model-based, and systematic. Random tools are the most straightforward; they randomly generate inputs to test Android apps. A widely used tool named Monkey [14] belongs to the random category. Random tools may generate a very large number of test cases. Model-based tools first construct a model (typically in the form of finite state machines) based on the GUI of an app; this model is then explored to create test cases. GUIRipper [7], PUMA [10] and A^{3}E-Depth-first [8] belong to this category. The final category (i.e., systematic) of tools uses often complicated and expensive techniques (e.g., symbolic execution, evolutionary algorithm) in an effort to more systematically generate test cases that can possibly achieve higher coverage. Among the six tools that Choudhary et al. have investigated, ACTeve [23] belongs to this category.

In this study, we include one random tool (i.e., Monkey [14]), and two model-based tools (i.e., GUIRipper [8] and PUMA [10]) from Choudhary *et al.*'s study. We exclude ACTEve, Dynodroid, and A3E-Depth-first because we cannot run them on our experimental machine due to compatibility issues. After Choudhary et al.'s study, additional test case generation tools are proposed in the literature. Therefore, we include two other tools, i.e., Droidmate [4] and Droidbot [15]. Simple descriptions of the selected tools are given below:

Monkey [14] generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. As Monkey has become a part of the Android developer toolkit, it is easy to install and use Monkey. Users need to configure by specifying the number of events they want Monkey to generate.

GUIRipper [7] uses a model-based exploration strategy to test apps and generates test cases in JUnit format. It builds a model of the app under test on the fly by collecting the information of the GUI of the app. Each state in the model keeps a list of events that can be generated. GUIRipper performs a depth-first search (DFS) procedure to generate input events from the model. During the DFS procedure, it restarts the exploration from the starting state when it cannot find new states. GUIRipper can only generate UI events but not system events; this may limit its ability to explore some app behaviors.

PUMA [10] is a generic framework that enables scalable and programmable UI automation. It supports the random exploration that is implemented by Monkey, as well as model-based exploration by providing a finite state machine (FSM) representation of an app under test and allowing users to modify the FSM and specify logic to generate events from the FSM. We use a version of PUMA packaged by Choudhary et al. [22] that is configured to generate test cases by exploring GUI models of apps.

Droidmate [4] implements an GUI-state based exploration strategy, which is inspired by Dynodroid [5]. The key idea of its exploration strategy is to interact with views (GUI elements) randomly, but give precedence to views that have been interacted with the least amount of times so far. Droidmate monitors sensitive APIs and user resources accessed by the app under test. During the exploration progress, Droidmate uses all the observed and monitored behavior of the app to decide which GUI element to interact with next or if the exploration is to be terminated. Droidmate terminates when a user-specified time limit is reached or when there are no views that can be interacted after two resets in a row. Droidmate needs to preprocess an app under test to make it *inlined*, that is, the app undergoes a slight Dalvik bytecode modification to enable Droidmate to monitor Android SDK's API calls.

Droidbot [15] dynamically builds a GUI model of an app under test by collecting GUI information and running process information. The model is a state transition graph, in which each node represents a device state, and each edge between two nodes represents the test case event that triggers the state transition. Droidbot uses a simple DFS procedure to generate test cases. Different from many other model-based tools, Droidbot is lightweight and does not require system modification or app instrumentation.

III. Experiment Setup

To validate the effectiveness of mining sandboxes, we run test case generation tools, construct sandboxes for benign apps based on sensitive APIs called in the execution of the generated test cases, and investigate the ability of those sandboxes to detect malicious behaviors in the malware that piggybacks the corresponding benign apps. Figure 2 presents the experiment setup process. The following subsections describe our app instrumentation strategy that we use to identify sensitive APIs that are called during test case execution, how we run the different test case generation tools, how we select the benign-malicious app pairs that we include in this study, and the details of our two experiments.

A. App Instrumentation

To collect API call traces of the selected apps when running test case generation tools, we use Droidfax proposed by Cai *et al.* [16]. Droidfax instruments the Android (Dalvik) bytecode of each app for API call profiling and inter-component communication (ICC) intent tracing using static program analysis. Droidfax uses Android logging utility and the logcat [4] tool to record API calls. Each API call recorded by Droidfax is in the format of *caller* → *callee*. Based on generated API call traces, we can build a *dynamic call graph*. Each node of the graph is an API (executed as a caller or callee), and each edge represents an API call which is the number of times the API is called in the traces.

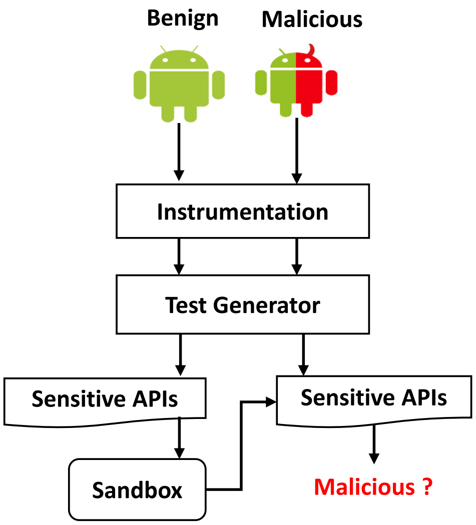

Fig. 2. Experiment Setup.

B. Running Automated Test Case Generation Tools

We use 5 automated test case generation tools in our experiment: Monkey, Droidmate, Droidbot, GUIRipper, and PUMA. Monkey is part of Android SDK, which requires no additional effort to install on our experiment environment. We download the source codes of Droidmate[5] and Droidbot[6] from their GitHub repositories then installed them according to instructions included in the repositories. We get GUIRipper and PUMA from the replication package released by Choudhary *et al.* [22] and modify their configuration to make them run in our experiment machine. As both Droidmate and PUMA work on Android SDK version 19, we configure the emulator to run Android SDK version 19. An exception is made for running GUIRipper; it is not open source and the version that we have based on Choudhary et al.'s study [22] only works for Android SDK version 10. Thus, for experiments involving GUIRipper, we configure our emulator to run Android SDK version 10. Our emulator is also configured to have 2GB of RAM with Intel x86 CPU architecture. The emulator in turn is run on a Mac OS 10.12 (Sierra) laptop running Intel Core i5 CPU (2.3 GHz).

C. App Selection

To investigate the effectiveness of mining sandboxes, we need pairs of benign app and malicious app that infects the benign app. The malicious apps used in our study are piggybacked apps, which are built by unpacking benign apps and grafting some malicious code to them. Previous studies shows most malware is piggybacked of benign apps, e.g., 80% of the malicious samples in the dataset MalGenome [18] are built through repackaging. Thus, we believe pairs of benign app and a malicious app that *piggybacks* it is a good dataset for our study. We used a piggybacked Android app dataset collected

[4] https://developer.android.com/studio/command-line/logcat.html

[5] https://github.com/konrad-jamrozik/droidmate

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

TABLE I
TEN PAIRS OF MALICIOUS-BENIGN APPS USED IN OUR FIRST EXPERIMENT.

Pair Index	Package	Category	Functionality
P1	com.google.android.diskusage	Tools	Find files and directories on storage card
P2	org.pyload.android.client	System	An Android client for pyload, which is a download manager written in Python
P3	com.chinat2t10513zw.templte	Lifestyle	Provides wedding ceremony information
P4	com.content.ugly.meter	Entertainment	Take faces of people and give a rating on ugly scale
P5	andrei.brusentcov.lnguagepazzle.en	Education	Learn English words by looking pictures then choosing corresponding characters
P6	com.northpark.beautycamera	Beauty	Take selfie photos and make the photo look more beautiful
P7	pl.netigen.bestbassguitarfree	Music & Audio	A guitar simulator
P8	oms.wmessage	Communication	Send text messaging and provide some Chinese text message templates
P9	cz.romario.opensudoku	Puzzle	An open source sudoku game.
P10	com.nesnet.android.cantonese	Book & Reference	A cantonese dictionary

by Li *et al.* [17], which contains 2,750 Android apps and 1,497 app pairs[7]. Each pair has one benign app and one malicious app, which is piggybacked on the benign app.

However, not all apps in the dataset can be used in our study. First, DROIDFAX cannot instrument some apps. Out of the 2,750 apps, only 844 can be successfully instrumented by DROIDFAX. Among these 844 apps, we fail to install a number of them on the emulator used in our study due to various compatibility issues with the SDK version and other settings of our emulators. These incompatibilities cause errors to be thrown or apps end gracefully right after they are started. After removing incompatible apps, we are left with 112 app pairs which we use for this study[8]

D. *Two Experiments:* `SmallE` *and* `LargeI`

We have two experiments in our study. In the first experiment (`SmallE`), to explore behavior of the app under test, we configure each automated test case generation tool to run for each individual app for one hour and repeat this process 5 times. We randomly select 10 app pairs for this experiment since it is not possible to run all 112 pairs of apps in limited time and resources (112 × 2 apps × 5 tools × 5 runs × 1 hour ≈ 233 days). Table I presents the information of these selected 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), categories of the app pairs (Category), and descriptions of the functionalities of the app pairs (Functionality). These 10 pairs of apps belong to different categories and have different functionalities. For example, both of P1 and P2 can access the storage of mobile devices since they need to manage files; P4 and P6 can take photos; P5 and P9 are game apps. In the second experiment (`LargeI`), we run the remaining 102 pairs but each automated test case generation tool is only allowed to run for one minute on each app.

IV. EXPERIMENT RESULTS

In this section, we first present the results of the first experiment (`SmallE`). Then, we present the results of the second experiment that runs inexpensive analysis on 102 app pairs (`LargeI`).

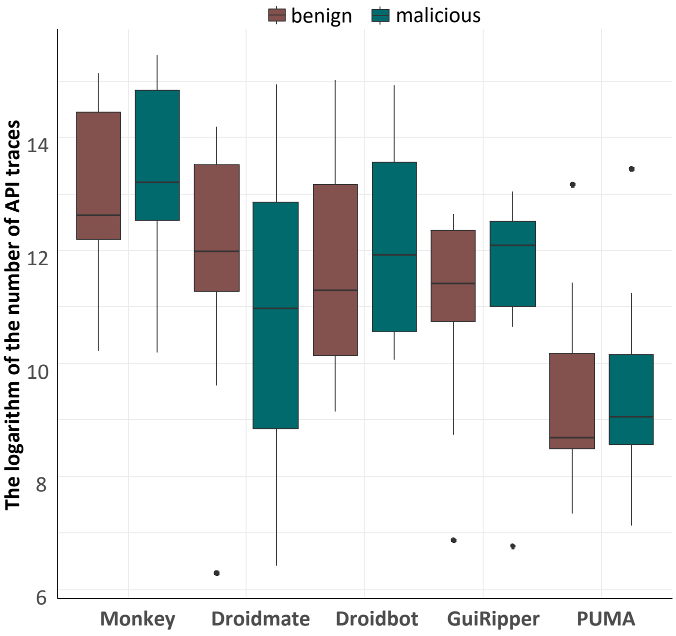

Fig. 3. Total Number of API Traces Generated by the Five Test Case Generation Tools.

A. *Experiment One:* `SmallE`

Statistics. We present some statistics based on the API traces generated by running test cases produced by the five test case generation tools. Figure 3 shows the number of API traces across the 10 pairs of apps under test for each tool[9]. The y-axis of this plot is the logarithm of the number of API traces as the range of the number of API traces is too large. Among the five test case generation tools, Monkey can generate the largest number of API traces in our experiment (i.e., more than one million API traces per run). This is because Monkey follows a random exploration strategy that might generate a large number of invalid inputs while all the other tools use model-based exploration strategy. PUMA has the smallest number of API

[7] All apps can be downloaded from Androzoo [24], which is a collection of Android applications collected from multiple markets.

[8] https://github.com/baolingfeng/SANER2018Sandboxes

[9] Note that the black dots in the boxplots correspond to outliers.

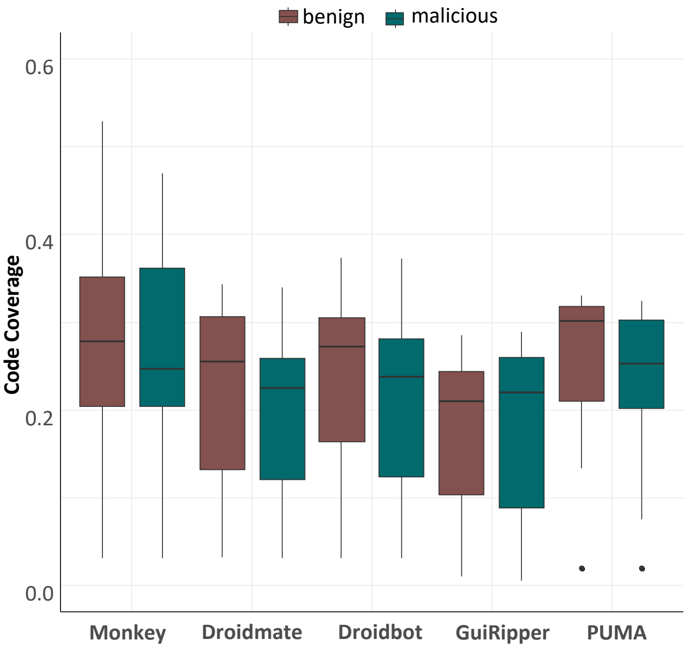

Fig. 4. Averaged Code Coverage of the Five Test Case Generation Tools.

traces and its average number of API traces is approximately 75,000. The numbers of generated API traces for different apps are also significantly different. For example, the number of API traces for apps in pair P4 is much smaller than that of other pairs.

We also measure the code coverage achieved by the test case generation tools. We focus on method-level code coverage, i.e., proportion of methods covered by the generated test cases. Figure 4 presents the average code coverage of the five test case generation tools respectively. Although Monkey uses a simple random exploration strategy to explore the behavior of the app under test, it achieves the highest code coverage (0.27 on average for both benign and malicious apps). All the other tools have similar code coverage. One reason for the low code coverage might be that many of the unexecuted parts of the app are third party library code. Li *et al.* reported that on average, 41% of an Android app code is contributed by third party libraries [25]. It is expected that these library code are not fully covered by the test cases because they are not fully used by the app. Moreover, Choudhary *et al.* [22] have also reported similar code coverage scores.

Effectiveness of Detecting Malicious Behavior. For each test case generation tool, given a pair of apps, we generate test cases and then we run them on the two apps. We next construct a sandbox based on sensitive APIs called by the benign app in the pair and check if this sandbox can identify the malicious app as such. The malicious app is detected by the sandbox if it calls other sensitive APIs but not called by the benign app. Figure 5 presents an example of the sensitive API calls identified from the API traces generated for the benign and malicious apps of pair P8. There is only one sensitive API call, i.e., call to “android.webkit.WebView.loadDataWithBaseURL(...)”, in the API traces of the benign app. However, three additional sensitive APIs exist in the API traces of the malicious app, which shows that the malicious app tries to run background and steal the device and subscriber ID.

Benign API

android.webkit.WebView.loadDataWithBaseURL(...)

Malicious APIs

android.webkit.WebView.loadDataWithBaseURL(...)
android.os.PowerManager$WakeLock.void acquire()
android.telephony.TelephonyManager.getDeviceId()
android.telephony.TelephonyManager.getSubscriberId()

Fig. 5. Sample Detected Malicious Behavior.

TABLE II
EFFECTIVENESS OF SANDBOXES CREATED BY RUNNING DIFFERENT TEST CASE GENERATION TOOLS.

Pair	Monkey	Droidmate	Droidbot	GuiRipper	PUMA
P1	√	⊗	⊗	⊗	⊗
P2	⊗	⊗	⊗	⊗	⊗
P3	√	√	√	√	√
P4	√	√	√	√	√
P5	⊗	⊗	⊗	√	⊗
P6	⊗	⊗	⊗	⊗	⊗
P7	√	√	√	√	√
P8	√	√	√	—	√
P9	√	√	√	√	√
P10	√	√	√	√	—

√ and ⊗ mean that whether or not the sandboxes created by running these tool detect the malicious apps, respectively. — means the test case generation tools fail to run the app.

Table II presents the results on whether the sandboxes constructed by running different test case generation tools can identify the malicious apps. All the sandboxes constructed by running these tools were able to identify the malicious behaviors for four pairs, i.e., P3, P4, P7, P9. For pair P8 and P10, there were four tools of which the corresponding sandboxes identified the malicious apps. GUIRipper and PUMA failed to run the apps of pair P8 and P10, respectively. The sandbox constructed by running the Monkey tool detected the largest number (7) of malicious apps. Only it found that the malicious app in pair P1, which invoked the sensitive API “android.webkit.WebView.loadDataWithBaseURL(...)” – see Table III. The sandboxes built by running Droidmate, Droidbot, and GUIRipper detected the same number of malicious apps (6). The malicious apps identified by sandboxes constructed by running Droidmate and Droidbot were the same. Only the sandbox built by running GUIRipper detected the malicious apps for the pair P5. It identified three additional sensitive APIs invoked by the malicious app – see Table III. The sandbox built by running PUMA identified the smallest number of malicious apps, i.e., 5. All the constructed sandboxes did not identify the malicious apps in pair P2 and P6. This might be because

TABLE III
Detected Sensitive API Calls that Break the Mined Sandboxes.

Sensitive API	App Pairs
android.os.PowerManager$WakeLock.acquire()	P5
android.telephony.TelephonyManager.getCellLocation()	P10
android.telephony.TelephonyManager.getDeviceId()	P3, P5, P10
android.telephony.TelephonyManager.getLine1Number()	P8, P9
android.telephony.TelephonyManager.getSimSerialNumber()	P8, P9
android.telephony.TelephonyManager.getSubscriberId()	P5, P10
android.webkit.WebView.loadDataWithBaseURL(...)	P1, P7
android.webkit.WebView.loadUrl(java.lang.String)	P4
java.net.URL.openConnection()	P9

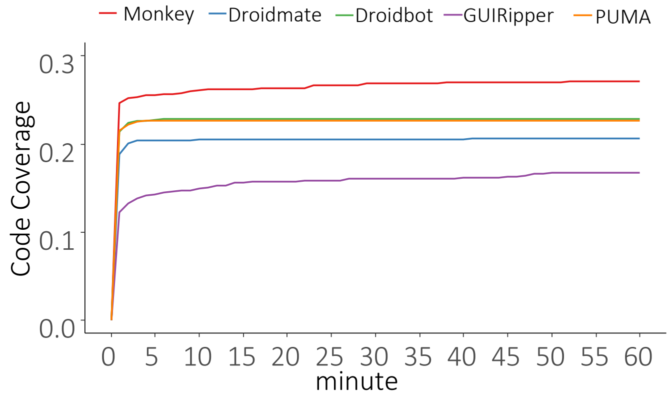

Fig. 6. Code Coverage of Test Cases Generated by the Five Tools over Time.

the malicious behavior of these two malicious apps were not covered by the generated test cases or the malicious behavior did not involve the use of additional sensitive APIs.

We also built a sandbox by combining all the five test case generation tools together. For each app pair, we used the set of sensitive APIs of the benign app detected by all the tools to build a sandbox. Then, we run all the tools for the malicious app and check if the sandbox could detect the malicious behavior. This sandbox can identify 8 out of 10 malicious apps except for the malicious apps in pair P2 and P6.

The sensitive APIs that differentiate the benign and malicious apps are presented in Table III. They can be divided into three categories: (1) power management (i.e., the method "Wakelock.acquire()"), (2) sensitive data access (i.e., the methods in class "android.telephony.TelephonyManager"), and (3) network connection (e.g., "java.net.URL.openConnection()"). These sensitive APIs are often used by malware to execute malicious operations; for example, a malware can use the APIs of class "Wakelock" to run in the background, and then access private data and send it over the network.

Mining sandbox method can effectively identify 8 out of the 10 malicious apps as such.

TABLE IV
Amount of Elapsed Time till Malicious Apps are Detected by Mined Sandboxes. "—" means that the corresponding tool is unable to detect additional sensitive APIs used by malware.

	Monkey	Droidmate	Droidbot	GUIRipper	PUMA
P1	3,412	—	—	—	—
P2	—	—	—	—	—
P3	1,642	8	9	35	23
P4	81	10	15	37	24
P5	—	—	—	13	—
P6	—	—	—	—	—
P7	44	13	24	368	26
P8	192	9	37	—	35
P9	352	56	10	33	22
P10	775	7	11	33	—
mean	928.3	17.2	17.7	86.5	26.0
std	1,138.6	17.5	10.0	126.1	4.7

Efficiency of Detecting Malicious Behavior. We also want to investigate how fast the sandboxes constructed by running these test case generation tools can identify the malicious behaviors. In the study of Jamrozik *et al.* [2], the sensitive APIs of apps used in the experiment can be called by the test case generation tool in several minutes [2]. Choudhary *et al.* [22] also reported that six test case generation tools evaluated in their study could hit the maximum coverage within a few minutes. Figure 6 presents code coverage of these five test case generation tools over time. The plot reports the mean coverage across all the 10 pairs of apps over 5 runs. The results are consistent with the results of Choudhary *et al.*'s study. All the tools achieve a coverage value that is close to the maximum value in one minute except GUIRipper. This is because GUIRipper frequently restarts the exploration from the starting state. This operation needs time to restart the emulator.

Next, we want to investigate the minimum amount of time the generated sandbox can identify malicious apps using test cases generated by the different test case generation tools. We show the result in Table IV which is the average minimum time of 5 runs. The symbol "–" in the table means that the tool cannot detect additional sensitive APIs used by the malware. There are no test case generation tools that detect malicious behaviors for pair P2 and P6. The last two columns correspond to the mean and standard deviation time across all detected malicious apps. Test cases generated using Monkey are the least efficient in detecting malicious apps. The shortest time it takes to flag a malicious app as such is one minute (P7), while the longest time is close to one hour (P1). This wide variation may be caused due to the random nature of Monkey. Interestingly, for the other tools that follow model-based exploration strategy, the variation in detection time is small. The average time to detect malicious apps using all other tools (except Monkey) is less than two minutes. The efficiency of test cases generated by Droidmate, Droidbot and PUMA are comparable (mean $<$ 1 minute), while those generated by GUIRipper require a bit more time to identify malicious behavior (mean $\approx$ 1.5 minutes).

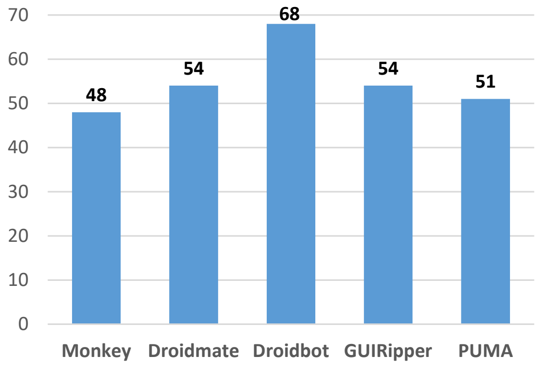

Fig. 7. Number of pairs of apps whose sets of sensitive APIs differ.

> Sandboxes constructed by running four of the five test case generation tools (except Monkey) can detect malicious behaviors in less than two minutes.

B. *Experiment Two:* `LargeI`

The result of the first experiment (see Figure 6) shows that almost all the tools achieve a coverage that is close to the maximum value in one minute. So, in this experiment, we only allow each tool to run on each individual app for one minute; this setting allows us to investigate the effectiveness of mining sandbox method on 102 pairs of apps.

Figure 7 presents the number of malicious apps detected by sandboxes constructed by running each tool. Sandboxes built by running Monkey, Droidmate, Droidbot, GUIRipper, and PUMA identified 48, 54, 68, 54, and 51 malicious apps among the 102 malicious apps, respectively. The sandboxes built by running Droidbot identified the largest number of malicious apps, which is much more than that of the sandboxes built by the other tools. The sandboxes built by running the other four tools have similar effectiveness – they detected 48-54 malicious apps. We also built sandboxes by combining multiple test case generation tools together. As the sandbox built by Droidbot had the best performance, we built sandboxes by combining Droidbot with each of the other four tools. We found that the number of identified malicious apps detected is increased to 73 (Droidbot+Monkey), 74 (Droidbot+Droidmate), 77 (Droidbot+GUIRipper), 71 (Droidbot+PUMA), respectively. We also combined all the five tools together. For this setting, we found that 77 out of the 102 malicious apps were identified.

The failure cases can be put into two groups: (1) the sets of sensitive APIs exercised by a test case generation tool for a benign app and its corresponding piggybacked malware are identical and non-empty, and (2) no sensitive APIs can be exercised by a test case generation tool. Table V presents the number of app pairs that fall into the two cases for each test case generation tool. We find that most failure cases belong to the first group.

TABLE V
FAILURE CASES.

	Identical Non-Empty Set of Sensitive APIs	No Sensitive APIs
Monkey	34	20
Droidmate	24	24
Droidbot	16	18
GUIRipper	34	14
PUMA	27	24

> Sandboxes generated by running automated test case generation tools for a short time (i.e., 1 minute) are able to detect 77 out of 102 malicious apps. The best test case generation tool is Droidbot. Still, by combining the tools together we can boost the performance further.

V. Discussion

In this section, we describe some implications and threats to validity of this work.

A. *Implications*

In the following paragraphs, we highlight some implications based on the findings of the study:

Mining sandboxes can effectively detect malicious behaviors. Our experiments show that 75.5%–77.2% of malware we investigated in this work can be detected by the sandboxes constructed by running the five selected test case generation tools. This complements the findings of Jamrozik et al. [2] that highlight there are only a few false alarms when using mining sandbox method to protect apps.

Multiple test case generation tools can be used to boost effectiveness of mined sandboxes. We found that the sandboxes built by the studied test case generation tools are capable of detecting different malicious behaviors for different apps. In the first experiment `SmallE`, the sandboxes inferred by running Monkey detects the largest number of malicious apps (i.e., 7 out of the 10 malicious apps) despite of its simple random exploration strategy. But the time it takes to detect the additional sensitive API calls in the malicious apps is much longer than that of the other tools. Additionally, if we combine all the tools together to build sandboxes, we can identify more malicious apps (i.e., 8 malicious apps). In the second experiment `LargeI`, the sandbox constructed by running Droidbot detects more malicious apps than the other tools (i.e., 71 *vs.* 54, 54, 61, and 58 for Monkey, Droidmate, GUIRipper, and PUMA, respectively). More importantly, we discover that by combining Droidbot with the other tools to construct a sandbox, the number of identified malicious apps increases. Therefore, it is better to use several test case generation tools together to build sandboxes.

More work is needed to further improve the effectiveness of mining sandboxes. These following directions seem promising:

1) Better automated test generation tools are required to cover more behavior of an app under test. Both Choudhary *et al.*'s study [22] and our study show that the code coverage of the automated test case generation tools is not high, and needs improvement.
2) As Android platform evolves rapidly, some tools are not compatible with latest Android versions. For example, Droidmate [4] only supports the Android SDK version 19 or 23. Thus, more work needs to be invested in maintaining automated test case generation tools so that they remain up-to-date and easy to be used by practitioners.
3) Our sandboxes are simple – they are only collections of allowable sensitive API calls. A number of malware that goes undetected in this work may perform malicious behaviors via a subset of sensitive APIs used by the benign app. Thus, in the future, it would be interesting to create more complex sandboxes that can capture additional constraints. For example, the sandboxes could include constraints between API calls specified in temporal logics, etc. These more sophisticated sandboxes could potentially identify more malware.
4) This work ignores parameter values. Future work can investigate possibility of mining important constraints governing parameter values of benign apps (e.g., by using Daikon [26]).

B. Threats to Validity

Internal Validity. One of threats to internal validity relates to implementation errors. We have carefully inspected our scripts to run the selected test case generation tools. However, still there could be errors that we do not detect. The randomness involved in the test case generation tools might be a threat to validity. To reduce this threat, we run each tool on each app 5 times and report the average effectiveness. In the second experiment, we only run the test case generation tools for one minute. It is possible that the coverage of the generated test cases has not converged yet. Still, our first experiment finds that the coverage of test cases generated by most tools reaches close to the maximum value within one minute.

Another threat to validity is that we did not reuse the whole implementation of Boxmate – we simply use its test case generation tool, i.e., Droidmate. Integrating each test case generation tool into Boxmate requires a lot of resource and time, which we leave for future work. Additionally, different from the original work by Jamrozik et al. that ignores *most* parameter values, in this work, we ignore *all* parameter values.

External Validity. Threats to external validity relates to the generalizability of our findings. We acknowledge the following threats:

1) In our first experiment, we only analyzed 10 pairs of APK files due to limited time and resources. Still, the number of apps considered is similar to those considered by many past studies that also perform dynamic analysis on Android apps [7], [23], [9]. Moreover, to mitigate this threat to external validity, we have performed a second experiment which includes 102 more app pairs. In the future, we plan to analyze more apps using the setting of the first experiment by devoting more resources and running the apps in parallel.
2) All the app pairs considered in this work are from piggybacked app dataset released by Li et al [17]. Piggybacked apps do not cover all categories of Android malware. Still, most malware is piggybacked of benign apps, e.g., 80% of the malicious samples in the dataset MalGenome [18] are built through repackaging.
3) We used 5 different test case generation tools to analyze the selected apps. There are a number of other test case generation tools that have been proposed in the literature (see Section VI-B). We have not considered these other tools.

In the future, we plan to reduce the threats to external validity by investigating more mobile applications as well as more automated test case generation tools from the industry and academia.

VI. Related Work

In this section, we highlight a number of previous research studies that are related to our work. In section VI-A, we discuss related works in sandbox mining. Next, Section VI-B describes state-of-the-art and popular test case generation techniques for Android apps. Then, we highlight works in adequacy of test case generations techniques Section VI-C.

A. Sandboxing

Our work extends the first sandbox mining paper by Jamrozik et al. [2]. While Jamrozik et al.'s have argued for the effectiveness of Boxmate and demonstrated its low false alarm rate, they have not evaluated it with real malware. This work validates the effectiveness of sandbox mining with real malware. Additionally, we investigated multiple test case generation tools in addition to the one investigated in their work. There are a number of other work on developing and analyzing sandboxes. For example, Cappos et al. proposed a more secure sandbox with a security layer that can prevent attackers from leveraging bugs in privileged functionalities [27]. Also, Graziano et al. 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 [28].

B. Automated Test Case Generation for Android

Recently, there are several tools that are proposed to generate test cases for mobile applications (or apps). There are three major behavior exploration strategies employed by automated test case generation approaches: random exploration (e.g., [14], [5], [10], [4]), model based exploration (e.g., [8], [9], [7], [29], [30], [15], [31]), and systematic exploration (e.g., [32], [12], [33]) strategies.

Monkey is a well-known testing tool that comes with Android Development Kit [14]. The tool is widely adopted as it is easy to use and highly compatible with different Android versions. Machiry et al. proposed Dynodroid generates relevant inputs to apps under test. Dynodroid leverages a novel "observe-select-execute" strategy to efficiently generate random events and select the ones related to current execution states of the apps [5]. Hao et al. proposed a novel tool, named PUMA, that makes UI automation programmable, and allow users to implement arbitrary dynamic analyses on Android applications [10]. Jamrozik et al. proposed Droidmate which is a fully automated GUI execution generator for Android applications [4]. Droidmate dynamically monitors sensitive APIs and resources assessed by an application, and decides which GUI elements to during exploration process for test case generation [4].

Azim et al. presented A^{3}E that systematically explores Android applications without assessing to their source code [8]. A^{3}E contains two distinct exploration strategies: targeted and depth-first exploration [8]. Choi et al. proposed a machine learning based approach, named SwiftHand, to actively infer finite-state machine based models of a GUI application [9]. Amalfitano et al. developed GUIRipper [7] that systematically explores GUIs of apps by maintaining state-machine models of GUIs, named GUI Tree models [7]. Amalfitano et al. extended AndroidRipper to MobiGUITAR by defining new test adequacy criteria that are based on state machines and providing fully automated testing that works with mobile platform security [29]. Yang et al. introduced ORBIT that performs static analysis on source code to extract actions associated with GUI states of Android applications [30]. Li et al. presented a light-weight UI-guided test case generator, named Droidbot, that supports model-based test case generation with minimal extra requirements and require no instrumentation [15]. Baek et al. proposed an automated model-based Android GUI testing framework, named GUICC, that supports multi-level GUI Comparison Criteria to construct accurate GUI models for test case generation [31].

Anand et al. presented a new technique, named ACTEve, that employs concolic execution for generating sequences of events for Android applications with available source code. Similarly, Jensen et al. applied concolic execution to generate sequences of user events that can reach to target states in an Android application [32]. Mahmood et al. introduced an evolutionary algorithm based testing framework, named EvoDroid, for generating relevant test cases for Android application [12]. Wong et al. proposed IntelliDroid that leverages both static and dynamic analyses to generate test cases for several Android dynamic analysis tools. [33].

C. Adequacy of Test Case Generation Techniques

Choudhary *et al.* evaluated the effectiveness of six test case generation tools for Android applications using different metrics [22]. According to their findings, the studied tools from academia are no better than Monkey when generating test cases for open-source apps [22]. Zeng et al. conducted an industrial case study by employing Monkey on WeChat as well as propose a new approach to improve the limitations of Monkey [34]. Gopinath et al. conducted an empirical study on hundreds of open-source projects from Github and assess quality of test cases given various coverage levels. In their study, they leveraged both of human generated test cases as well as Randoop [35] generated test cases [36]. According to Gopinath et al.'s findings, statement coverage is a good indicator of test suite effectiveness [36]. Inozemtseva et al. generated 31,000 test suites for five large-scale software systems that contain up to 724,000 lines of source code, and, importantly, discovered that test coverage is not a good indicator of test suite effectiveness [37]. Kochhar et al. analyzed Apache HTTPClient and Mozilla Rhino to understand the correlation between the test suite coverage, size and effectiveness [38]. Zhang et al. analyzed five large-scale open-source projects to investigate the relationship between test suite effectiveness and number of assertions, types of assertions and assertion coverage [39].

VII. Conclusion and Future Work

In this paper, we investigate the effectiveness of mining sandboxes on detecting malicious apps using five test case generation tools. We make use of pairs of malware and benign app it infects to investigate whether the sandbox built based on sensitive APIs called by the benign app can detect the malicious behavior in the corresponding malware effectively. We conduct two experiments. In the first experiment, we select 10 pairs of apps and allow test case generation tools to run for one hour; while in the second experiment, we select 102 pairs of apps and allow these tools to run for one minute. The results of the first experiment show that the sandbox constructed by combining all the five test case generation tools can identify 8 out of 10 malicious apps; while sandboxes built by running Monkey, Droidmate, Droidbot, GUIRipper, and PUMA can detect 7, 6, 6, 6, 5 malicious apps, respectively. In the second experiment, 75.5% (77 out of 102) of malicious apps can be identified by the sandbox constructed by combining all the five test case generation tools. The best test case generation tool is Droidbot. The performance can also be boosted by combing the other tools together.

As future work, we plan to expand our study to better address the threats to internal and external validity. We also plan to build better sandboxes by designing improved test case generation tools and inferring more sophisticated models of benign behaviors.

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 *Proceedings of the 23rd international conference on World wide web*. ACM, 2014, pp. 39–50.

[2] 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*, 2016, pp. 37–48.

[3] P. Xiao, A. Pan, L. Long, and Y. Song, “What can you do to an APK without its private key except repacking?” in *BlackHat Mobile Security Summit*, 2015.

[4] 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*, 2016, pp. 293–294.

[5] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An input generation system for android apps,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[6] R. Sasnauskas and J. Regehr, “Intent fuzzer: crafting intents of death,” in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. ACM, 2014, pp. 1–5.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, “Using gui ripping for automated testing of android applications,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[8] 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.

[9] W. Choi, G. Necula, and K. Sen, “Guided gui testing of android apps with minimal restart and approximate learning,” in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.

[10] S. Hao, B. Liu, S. Nath, W. G. 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*. ACM, 2014, pp. 204–217.

[11] S. Anand, M. Naik, M. J. Harrold, and H. Yang, “Automated concolic testing of smartphone apps,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 59.

[12] R. Mahmood, N. Mirzaei, and S. Malek, “Evodroid: Segmented evolutionary testing of android apps,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.

[13] H. van der Merwe, B. van der Merwe, and W. Visser, “Execution and property specifications for jpf-android,” *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–5, 2014.

[14] “https://developer.android.com/studio/test/monkey.html,” accessed: October 2017.

[15] 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, pp. 23–26.

[16] H. Cai and B. Ryder, “Understanding application behaviours for android security: A systematic characterization,” *Computer Science Technical Reports (2016). https://vtechworks. lib. vt. edu/bitstream/handle/10919/71678/cairyder_techreport. pdf*, 2016.

[17] 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.

[18] 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.

[19] 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.

[20] 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*, 2011, pp. 627–638.

[21] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “Pscout: analyzing the android permission specification,” in *the ACM Conference on Computer and Communications Security, CCS’12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 217–228.

[22] S. R. Choudhary, A. Gorla, and A. Orso, “Automated test input generation for android: Are we there yet?(e),” in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 429–440.

[23] S. Anand, M. Naik, M. J. Harrold, and H. Yang, “Automated concolic testing of smartphone apps,” in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE’12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 59. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[24] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Androzoo: Collecting millions of android apps for the research community,” in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 468–471.

[25] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, “An investigation into the use of common libraries in android apps,” in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 403–414.

[26] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The daikon system for dynamic detection of likely invariants,” *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[27] 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 *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 1057–1072.

[28] 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 *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, 2010, pp. 212–223.

[29] 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.

[30] W. Yang, M. R. Prasad, and T. Xie, “A grey-box approach for automated gui-model generation of mobile applications,” in *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, 2013, pp. 250–265.

[31] Y. M. Baek and D. Bae, “Automated model-based android GUI testing using multi-level GUI comparison criteria,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 238–249.

[32] C. S. Jensen, M. R. Prasad, and A. Møller, “Automated testing with targeted event sequence generation,” in *International Symposium on Software Testing and Analysis, ISSTA ’13, Lugano, Switzerland, July 15-20, 2013*, 2013, pp. 67–77.

[33] M. Y. Wong and D. Lie, “Intellidroid: A targeted input generator for the dynamic analysis of android malware,” in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[34] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, “Automated test input generation for android: Are we really there yet in an industrial case?” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 987–992.

[35] C. Pacheco and M. D. Ernst, “Randoop: Feedback-directed random testing for java,” in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA)*, 2007, pp. 815–816.

[36] R. Gopinath, C. Jensen, and A. Groce, “Code coverage for suite evaluation by developers,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 72–82.

[37] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 435–445.

[38] P. S. Kochhar, F. Thung, and D. Lo, “Code coverage and test suite effectiveness: Empirical study with real bugs in large systems,” in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, 2015, pp. 560–564.

[39] Y. Zhang and A. Mesbah, “Assertions are strongly correlated with test suite effectiveness,” in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 214–224.