2015 30th IEEE/ACM International Conference on Automated Software Engineering

Automated Test Input Generation for Android: Are We There Yet?

Shauvik Roy Choudhary
Georgia Institute of Technology, USA
Email: shauvik@cc.gatech.edu

Alessandra Gorla
IMDEA Software Institute, Spain
Email: alessandra.gorla@imdea.org

Alessandro Orso
Georgia Institute of Technology, USA
Email: orso@cc.gatech.edu

***Abstract*—Like all software, mobile applications ("apps") must be adequately tested to gain confidence that they behave correctly. Therefore, in recent years, researchers and practitioners alike have begun to investigate ways to automate apps testing. In particular, because of Android's open source nature and its large share of the market, a great deal of research has been performed on input generation techniques for apps that run on the Android operating systems. At this point in time, there are in fact a number of such techniques in the literature, which differ in the way they generate inputs, the strategy they use to explore the behavior of the app under test, and the specific heuristics they use. To better understand the strengths and weaknesses of these existing approaches, and get general insight on ways they could be made more effective, in this paper we perform a thorough comparison of the main existing test input generation tools for Android. In our comparison, we evaluate the effectiveness of these tools, and their corresponding techniques, according to four metrics: ease of use, ability to work on multiple platforms, code coverage, and ability to detect faults. Our results provide a clear picture of the state of the art in input generation for Android apps and identify future research directions that, if suitably investigated, could lead to more effective and efficient testing tools for Android.**

I. Introduction

In the past few years, we have witnessed an incredible growth of the mobile applications (or simply, "apps") business. Apps, like all software, must be adequately tested to gain confidence that they behave correctly. It is therefore not surprising that, with such a growth, the demand for tools for automatically testing mobile apps has also grown, and with it the amount of research in this area. Most of the researchers' and practitioners' efforts in this area target the Android platform, for multiple reasons. First and foremost, Android has the largest share of the mobile market at the moment, which makes this platform extremely appealing for industry practitioners. Second, due to the fragmentation of devices and OS releases, Android apps often suffer from cross-platform and cross-version incompatibilities, which makes manual testing of these apps particularly expensive—and thus, particularly worth automating. Third, the open-source nature of the Android platform and its related technologies makes it a more suitable target for academic researchers, who can get complete access to both the apps and the underlying operating system. In addition, Android apps are developed in Java. Even if they are compiled into Dalvik bytecode, which significantly differs from Java bytecode, there exist multiple frameworks that can transform Dalvik bytecode into formats that are more familiar and more amenable to analysis and instrumentation (*e.g.,* Java Bytecode [26], Jimple [7], and smali [29]). For all these reasons, there has been a great deal of research in static analysis and testing of Android apps. In the area of testing, in particular, researchers have developed techniques and tools to target one of the most expensive software testing activities: test input generation. There are in fact a number of these techniques in the literature nowadays, which differ in the way they generate inputs, the strategy they use to explore the behavior of the app under test, and the specific heuristics they use. It is however still unclear what are the strengths and weaknesses of these different approaches, how effective they are in general and with respect to one another, and if and how they could be improved.

To answer these questions, in this paper we present a comparative study of the main existing *tool-supported* test input generation techniques for Android.[1] The goal of the study is twofold. The first goal is to assess these techniques (and corresponding tools) to understand how they compare to one another and which ones may be more suitable in which context (*e.g.,* type of apps). Our second goal is to get a better understanding of the general tradeoffs involved in test input generation for Android and identify ways in which existing techniques can be improved or new techniques be defined. In our comparison, we ran the tools considered on over 60 real-world apps, while evaluating their usefulness along several dimensions: *ease of use*, ability to work on *multiple platforms*, *code coverage*, and *fault detection*. We evaluated the ease of use of the tools by assessing how difficult it was to install and run them and the amount of manual work involved in their use. Although this is a very practical aspect, and one that normally receives only limited attention in research prototypes, (reasonable) ease of use can enable replication studies and allow other researchers to build on the existing technique and tool. Because of the fragmentation of the Android ecosystem, another important characteristic for the tools considered is their ability to work on different hardware and software configurations. We therefore considered and assessed also this aspect of the tools, by running them on different versions of the Android environment. We considered coverage because test input generation tools should be able to explore as much behavior of the app under test as possible, and code coverage is typically used as a proxy for that. We therefore used the tools to generate test inputs for each of the apps considered and measured the coverage achieved by the different tools on each app. Although code coverage is a well understood and commonly used measure, it is normally a gross approximation of behavior. Ultimately, test input generation tools should generate inputs that are effective at revealing faults in the code under test. For this reason, in our study we also

[1] As discussed in Section III, we had to exclude some tools from our study.

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.89

measured how many of the inputs generated by a tool resulted in one or more failures (identified as uncaught exceptions) in the apps considered. We also performed additional manual and automated checks to make sure that the thrown exceptions represented actual failures.

Our results show that, although the existing techniques and tools we studied are effective, they also have weaknesses and limitations, and there is room for improvement. In our analysis of the results, we discuss such limitations and identify future research directions that, if suitably investigated, could lead to more effective and efficient testing tools for Android. To allow other researchers to replicate our studies and build on our work, we made all of our experimental infrastructure and data publicly available at http://www.cc.gatech.edu/~orso/software/androtest.

The main contributions of this paper are:

- A survey of the main existing tool-supported test input generation techniques for Android apps.
- An extensive comparative study of such techniques and tools performed on over 60 real-world apps.
- An analysis of the results that discusses strengths and weaknesses of the different techniques considered and highlights possible future directions in the area.
- A set of artifacts, consisting of experimental infrastructure as well as data, that are freely available and allow for replicating our work and building on it.

The remainder of the paper is structured as follows. Section II provides background information on Android. Section III discusses the test input generation techniques and tools that we consider in our study. Section IV describes our study setup and presents our results. Section V analyzes and discusses our findings. Finally, Section VI concludes the paper.

II. The Android Platform and Apps

Android apps run on top of a stack of three other main software layers. The *Android framework* provides an API such that apps can access facilities without dealing with the low level details of the operating system. So far, there have been 20 different framework releases and consequent changes in the API. Framework versioning is the first element that causes the fragmentation problem in Android. Since it takes several months for a new framework release to become predominant on Android devices, most of the devices in the field run older versions of the framework. Android developers thus constantly have to make an effort to make their apps compatible with older framework versions.

At runtime, the Zygote daemon creates a separate Dalvik Virtual Machine (Dalvik VM) for each running app, where a Dalvik VM is a register-based VM that can interpret Dalvik bytecode. The most recent version of Android includes radical changes in the runtime layer, as it introduces ART (*i.e.,* Android Run-Time), a new runtime environment that dramatically improves app performance and will eventually replace the Dalvik VM. The custom *Linux kernel*, which stands at the bottom of the Android software stack, provides the main functionality of the system. A set of *native code libraries*, such as WebKit, libc and SSL, communicate directly with the kernel and provide basic hardware abstractions to the runtime layer.

Android apps are mainly written in Java, although it is often the case that developers include native code to improve their performance. Java source code first gets compiled into Java bytecode, then translated into Dalvik bytecode, and finally stored into a machine executable file in .dex format. Apps are finally distributed in the form of apk files, which are compressed folders containing dex files, native code (whenever present), and other application resources. Android apps declare in the AndroidManifest.xml file their main components, which can be of four different types:

Activities are the components in charge of an app's user interface. Each activity is a window containing various UI elements, such as buttons and text areas. Developers can control the behavior of each activity by implementing appropriate callbacks for each life-cycle phase (*i.e.,* created, paused, resumed, and destroyed). Activities react to user input events such as clicks, and consequently are the primary target of testing tools for Android.

Services are application components that can perform long-running operations in the background. Unlike activities, they do not provide a user interface, and consequently they are usually not a direct target of Android testing tools, although they might be indirectly tested through some activities.

Broadcast Receivers and Intents allow inter-process communication. Apps can register broadcast receivers to be notified, by means of intents, about specific system events. Apps can thus, for instance, react whenever a new SMS is received, a new connection is available, or a new call is being made. Broadcast receivers can either be declared in the manifest file or at runtime, in the app's code. In order to properly explore the behavior of an app, testing tools should be aware of what are the relevant broadcast receivers, so that they could trigger the right intents.

Content Providers act as a structured interface to shared data stores, such as contacts and calendar databases. Apps may have their own content providers and may make them available to other apps. Like all software, the behavior of an app may depend on the state of such content providers (*e.g.,* on whether a list of contacts is empty or whether it contains duplicates). As a consequence, testing tools should "mock" content providers in an attempt to make tests deterministic and achieve higher coverage of an app's behavior.

Despite being GUI-based and mainly written in Java, Android apps significantly differ from Java standalone GUI applications and manifest somehow different kinds of bugs [14], [15]. Existing test input generation tools for Java [12], [21], [22] cannot therefore be straightforwardly used to test Android apps, and custom tools must be created instead. For this reason, a great deal of research has been performed in this area, and several test input generation techniques and tools for Android apps have been proposed. The next section provides an overview of the main existing tools in this arena.

III. Existing Android Testing Tools: An Overview

The primary goal of input generation tools for Android is to detect existing faults in apps under test. App developers are thus typically the main stakeholders for these tools, as by using the tools they can automatically test their apps and fix discovered issues before deploying them. The dynamic traces

TABLE I. OVERVIEW OF INPUT GENERATION TOOLS FOR ANDROID APPS. THE ROWS FOR TOOLS CONSIDERED IN OUR STUDY ARE HIGHLIGHTED.

Name	Available	Instrumentation		Events		Exploration strategy	Needs source code	Testing strategy
		Platform	App	UI	System			
Monkey [24]	✓	×	×	✓	×	Random	×	Black-box
Dynodroid [18]	✓	✓	×	✓	✓	Random	×	Black-box
DroidFuzzer [34]	✓	×	×	×	×	Random	×	Black-box
IntentFuzzer [28]	✓	×	×	×	×	Random	×	White-box
Null IntentFuzzer [25]	✓	×	×	×	×	Random	×	Black-box
GUIRipper [1]	✓[a]	×	✓	✓	×	Model-based	×	Black-box
ORBIT [33]	×	?	?	✓	×	Model-based	✓	Grey-box
A^{3}E-Depth-first [6]	✓	×	✓	✓	×	Model-based	×	Black-box
SwiftHand [8]	✓	×	✓	✓	×	Model-based	×	Black-box
PUMA [13]	✓	×	✓	✓	×	Model-based	×	Black-box
A^{3}E-Targeted [6]	×	×	✓	✓	×	Systematic	×	Grey-box
EvoDroid [19]	×	×	✓	✓	×	Systematic	×	White-box
ACTEve [3]	✓	✓	✓	✓	✓	Systematic	✓	White-box
JPF-Android [30]	✓	×	×	✓	×	Systematic	✓	White-box

a) Not open source at the time of this study.

generated by these tools, however, can also be the starting point of more specific analyses, which can be of primary interest to Android market maintainers and final users. In fact, Android apps heavily use features, such as native code, reflection and code obfuscation, that hit the limitations of almost every static analysis tool [4], [5], [11]. Thus, to explore the behavior of Android apps and overcome such limitations, it is common practice to resort to dynamic analysis and use test input generation tools to explore enough behaviors for the analysis [10], [32]. Google, for instance, is known to run every app on its cloud infrastructure to simulate how it might work on user devices and look for malicious behavior [17].

Test input generation tools can either analyze the app in isolation or focus on the interaction between the app and other apps and/or the underlying framework. Whatever is the final usage of these tools, the challenge is to generate relevant inputs to exercise as much behavior of the apps under test as possible. As Android apps are event-driven, inputs are normally in the form of events, which can either mimic user interactions (*UI events*), such as clicks, scrolls, and text inputs, or *system events*, such as the notification of a newly received SMS. Testing tools can generate such inputs following different strategies. They can generate them *randomly* or by following a *systematic* exploration strategy. In this latter case, exploration can either be guided by a *model* of the app, which can be constructed statically or dynamically, or exploit techniques that aim to achieve as much code coverage as possible. Along a different dimension, testing tools can generate events by considering Android apps as either a *black box* or a *white box*. In this latter case, they would consider the code structure. *Grey box* approaches are also possible, which typically extract high-level properties of the app, such as the list of activities and the list of UI elements contained in each activity, in order to generate events that will likely expose unexplored behavior.

Table I provides an overview of the existing test input generation tools for Android presented in the literature. To the best of our knowledge, this list is complete. For our study, we selected, among these tools, those that were available and had as their main objective to cover the state space of the app under test. We therefore ignored tools that only focus on making an app crash (e.g., intent fuzzers) and tools whose goal is to identify specific issues (e.g., deadlock detectors). The table reports all these tools, highlights the ones we considered in our study, and classifies them according to their characteristics. In particular, the table reports whether a tool is *publicly available*, distributed under restricted policies, or only presented in a paper, whether it *requires the source code* of the app under test, and whether it requires *instrumentation*, either of the app itself or of the underlying Android framework. The following sections provide more details on each of these tools, grouped based on their exploration strategy, and explain why we could not consider some of them in our study.

A. Random Exploration Strategy

The first class of test input generation tools we consider employs a random strategy to generate inputs for Android apps. In its simplest form, a random strategy generates only UI events; randomly generating system events would be highly inefficient, as there are too many such events, and apps usually react to only a few of them, and only under specific conditions.

The advantage of input generators based on a random exploration strategy is that they can efficiently generate events, and this makes them particularly suitable for stress testing. Their main drawback is that a random strategy would hardly be able to generate highly specific inputs. Moreover, these tools are not aware of how much behavior of the app has been already covered and are thus likely to generate redundant events that do not help the exploration. Finally, they do not have a stopping criterion that indicates the success of the exploration, but rather resort to a manually specified timeout.

Monkey [24] is the most frequently used tool to test Android apps, partly because it is part of the Android developers toolkit and does not require any additional installation effort. Monkey implements the most basic random strategy, as it considers the app under test a black-box and can only generate UI events. Users have to specify the number of events they want Monkey to generate. Once this upper bound has been reached, Monkey stops.

Dynodroid [18] is also based on random exploration, but it has several features that make its exploration more efficient compared to Monkey. First of all, it can generate system events,

and it does so by checking which ones are relevant for the app. Dynodroid gets this information by monitoring when an app registers a listener within the Android framework. For this reason it needs to instrument the framework. The random event generation strategy of Dynodroid is smarter than the one that Monkey implements. It can either select the events that have been least frequently selected (*Frequency* strategy) and can keep into account the context (*BiasedRandom* strategy), that is, events that are relevant in more contexts will be selected more often. For our study, we only considered the *BiasedRandom* strategy, which is the default one. An additional improvement of Dynodroid is the ability to let users manually provide inputs (*e.g.,* for authentication) when the exploration is stalling.

Most of the other tools that fall into this category are input fuzzers that aim to test inter-app communication by randomly generating values for intents. These tools are test input generators with a very specific purpose: they mainly aim to generate invalid inputs that make the app under test crash, thus testing the robustness of the app, rather than trying to cover its behavior. These fuzzers are also quite effective at revealing security vulnerabilities, such as denial-of-service vulnerabilities. Given the different purpose of these tools, we decided to exclude them from our study. In particular, we excluded the following three tools.

Null intent fuzzer [25] is an open-source basic intent fuzzer that aims to reveal crashes of activities that do not properly check input intents. While quite effective at revealing this type of problems, it is fairly specialized and not effective at detecting other issues.

Intent Fuzzer [28] mainly tests how an app can interact with other apps installed on the same device. It includes a static analysis component, built on top of FlowDroid [4], for identifying the expected structure of intents, so that the fuzzer can generate them accordingly. This tool has shown to be effective at revealing security issues. Maji et al. worked on a similar intent fuzzer [20], but their tool has more limitations than Intent Fuzzer.

DroidFuzzer [34] is different from other tools that mainly generate UI events or intents. It solely generates inputs for activities that accept MIME data types such as AVI, MP3, and HTML files. The authors of the paper show how this tool could make some video player apps crash. DroidFuzzer is supposed to be implemented as an Android app. However, it is not available, and the authors did not reply to our request for the tool.

B. Model-Based Exploration Strategy

Following the example of several Web crawlers [9], [23], [27] and GUI testing tools for stand alone applications [12], [21], [22], some Android testing tools build and use a GUI model of the app to generate events and systematically explore the behavior of the app. These models are usually finite state machines that have activities as states and events as transitions. Some tools build precise models by differentiating activity states (*e.g.,* the same activity with a button enabled and disabled would be represented as two separate states). Most tools build such model dynamically and terminate when all the events that can be triggered from all the discovered states lead to already explored states.

Using a model of the app should intuitively lead to more effective results in terms of code coverage, as it can limit the number of redundant inputs that a random approach generates. The main limitation of these tools stands in the state representation they use, as they all represent new states only when some event triggers changes in the GUI. Some events, however, may change the internal state of the app without affecting the GUI. In such situations, these algorithm would miss the change, consider the event irrelevant, and continue the exploration in a different direction. A common scenario in which this problem occurs is in the presence of services, as services do not have any user interface (see Section II).

GUIRipper [1], which later became MobiGUITAR [2], dynamically builds a model of the app under test by crawling it from a starting state. When visiting a new state, it keeps a list of events that can be generated on the current state of the activity and systematically triggers them. GUIRipper implements a DFS strategy and restarts the exploration from the starting state when it cannot detect new states during the exploration. It generates only UI events, thus it cannot expose behavior of the app that depends on system events. GUIRipper has two characteristics that make it unique among model-based tools. First, it allows for exploring an app from different starting states (although not in an automated fashion.) Second, it allows testers to provide a set of input values that can be used during the exploration. GUIRipper is publicly available and is now open source under the name AndroidRipper (https://github.com/reverse-unina/AndroidRipper). At the time of our comparative study, however, it was only available (close source) on Windows. We managed to include it in the study by porting the Windows scripts that interact with the Java core system to the Linux platform.

ORBIT [33] implements the same exploration strategy of GUIRipper, but statically analyzes the app's source code to understand which UI events are relevant for a specific activity. It is thus supposed to be more efficient than GUIRipper, as it should generate only relevant inputs. However, the tool is unfortunately not available, as it is propriety of Fujitsu Labs. It is unclear whether ORBIT requires any instrumentation of the platform or of the app to run, but we believe that this is not the case.

A^{3}E-Depth-First [6]: A^{3}E is an open source tool that implements two distinct and complementary strategies. The first one performs a depth first search on the dynamic model of the app. (In essence, it implements the exact same exploration strategy of the previous tools.) Its dynamic model representation is more abstract than the one used by other tools, as it represents each activity as a single state, without considering different states of the elements of the activity. This abstraction may lead to missing some behavior that would be easy to exercise if a more accurate model were to be used. We discuss the second strategy of A^{3}E, A^{3}E-Targeted, in Section III-C.

SwiftHand [8] aims to maximize the coverage of the app under test. Similarly to the previous tools, it uses a dynamic finite state machine model of the app, and one of its main characteristics is to optimize the exploration strategy to minimize the restarts of the app while crawling. SwiftHand generates only touching and scrolling UI events and cannot generate system events.

PUMA [13] includes a generic UI automator that implements the same basic random exploration as Monkey. The novelty of this tool is, in fact, not in its exploration strategy, but rather in its design. PUMA is a framework that can be easily extended to implement any dynamic analysis on Android apps based on its basic exploration strategy. Moreover, PUMA also allows for (1) implementing different exploration strategies, as the framework provides a finite state machine representation of the app, and (2) redefining the state representation and the logic to generate events. PUMA is publicly available and open source. It is, however, only compatible with the most recent releases of the Android framework.

C. Systematic Exploration Strategy

Some application behavior can only be revealed upon providing specific inputs. This is the reason why some Android testing tools use more sophisticated techniques, such as symbolic execution and evolutionary algorithms, to guide the exploration towards previously uncovered code. Implementing a systematic strategy leads to clear benefits in exploring behavior that would be hard to reach with random techniques. Compared to random techniques, however, these tools are considerably less scalable.

A^{3}E-Targeted [6] provides an alternative exploration strategy that complements the one described in Section III-B. The targeted approach relies on a component that, by means of taint analysis, can build the Static Activity Transition Graph of the app. Such graph is an alternative to the dynamic finite state machine model used by A^{3}E-Depth-First and allows the tool to cover activities more efficiently by generating intents. While A^{3}E is available on a public repository, this strategy does not seems to be, so we could not include it in our study.

EvoDroid [19] relies on evolutionary algorithms to generate relevant inputs. In the evolutionary algorithms framework, EvoDroid represents individuals as sequences of test inputs and implements the fitness function so as to maximize coverage. EvoDroid used to be publicly available on its project website, and we tried to install and run it. We also contacted the authors after we ran into some problems with missing dependencies, but despite their willingness to help, we never managed to get all the files we needed and the tool to work. Obtaining the source code and fixing the issues ourselves was unfortunately not an option, due to the contractual agreements with their funding agencies. Moreover, at the time of this writing, the tool is no longer available, even as a closed-source package.

ACTEve [3] is a concolic-testing tool that symbolically tracks events from the point in the framework where they are generated up to the point where they are handled in the app. For this reasons, ACTEve needs to instrument both the framework and the app under test. ACTEve handles both system and UI events.

JPF-Android [31] extends Java PathFinder (JPF), a popular model checking tool for Java, to support Android apps. This would allow to verify apps against specific properties. Liu and colleagues were the first who investigated the possibility of extending JPF to work with Android apps [16]. What they present, however, is mainly a feasibility study. They themselves admit that developing the necessary components would require much additional engineering efforts. Van Der Merwe and colleagues went beyond that and properly implemented and open sourced the necessary extensions to use JPF with Android. JPF-Android aims to explore all paths in an Android app and can identify deadlocks and runtime exceptions. The tool, however, requires its users to manually specify the sequence of input events to be used for the exploration. For this reason, the tool cannot be automatically run on an app, and we thus decided to exclude it from our study.

IV. Empirical Study

To evaluate the test input generation tools that we considered (highlighted in Table I), we deployed them along with a group of Android apps on a common virtualized infrastructure. Such infrastructure aims to ease tool comparisons, and we made it available so that researchers and practitioners can more easily evaluate new Android testing tools against existing ones. Our study evaluated tools according to four main criteria:

C1: Ease of use. We believe that usability should be a primary concern for all tool developers, even when tools are just research prototypes, as it highly affects reuse, research collaboration, and ultimately research impact. We evaluated the usability of each tool by considering how much effort it took us to install and use it.

C2: Android framework compatibility. One of the major problems in the Android ecosystem is fragmentation. Test input generation tools for Android should therefore ideally run on multiple versions of the Android framework, so that developers could assess how their app behaves in different environments.

C3: Code coverage achieved. The inputs that these tools generate should ideally cover as much behavior as possible of the app under test. Since *code coverage* is a commonly used proxy for behavior coverage, we measured the statement coverage that each tool achieved on each benchmark and then compared the results of the different tools.

C4: Fault detection ability. The primary goal of test input generators is to expose existing faults. We therefore assessed, for each tool, how many failures it triggered for each app. We then compared the effectiveness of different tools in terms of failure detection.

Each of these research questions is addressed separately in Sections IV-B (C1), IV-B (C2), IV-B (C3), and IV-B (C4).

A. Mobile App Benchmarks

We selected a common set of benchmarks to evaluate the tools. Since some of these tools are not maintained, and therefore may not be able to handle apps that utilize cutting-edge features, for our experiments we combined all the open source mobile app benchmarks that were used in the evaluation of at least one considered tool. We retrieved the same version of the benchmarks as they were reported in each paper. PUMA and A^{3}E were originally evaluated on a set of apps downloaded from the Google Play market. We excluded these apps from our dataset because some tools need the app source code, and therefore it would have been impossible to run them on these benchmarks.

We collected *68 apps* in total. Among those, 50 are from the Dynodroid paper [18], 3 from GUIRipper [1], 5 from ACTEve [3], and 10 from SwiftHand [8]. We are aware that the number of benchmarks is quite unbalanced in favor of

Dynodroid, and this may represent a threat to the validity of our study. However, such apps were randomly selected for the original Dynodroid study, and are quite diverse, which should alleviate such threat. Table II reports the whole list of apps that we collected, together with the corresponding version and category. For each app we report whether it was part of the original evaluation benchmarks for a specific tool and whether, during our evaluation, the tool crashed when attempting to exercise the app.

B. Experimental Setup

We ran our experiments on a set of Ubuntu 14.04 virtual machines running on a Linux server. We used `Oracle VirtualBox` (http://virtualbox.org) as our virtualization software and `vagrant` (http://vagrantup.com) to manage these virtual machines. Each virtual machine was configured with 2 cores and 6GB of RAM. Inside the virtual machine, we installed the test input generation tools, the Android apps, and three versions of the Android SDK: Versions 10 (Gingerbread), 16 (Ice-cream sandwich), and 19 (Kitkat). We chose these versions mainly to satisfy tool dependencies, and we selected the most recent and most popular at the time of the experiment. We evaluated each tool on the SDK version that it officially supported and then tried to run it on all three versions to assess framework compatibility. The emulator was configured to use 4GB of RAM, and each tool was allowed to run for 1 hour on each benchmark app. For every run, our infrastructure created a new emulator with necessary tool configuration to avoid side-effects between tools and apps. Given that many testing tools and apps are non-deterministic, we repeated each experiment 10 times and computed the mean values across all runs. Finally, to avoid bias, we ran each tool with its default configuration, that is, without tuning any available parameter.

For each run, we collected the code coverage for the app under test with `Emma` (http://emma.sourceforge.net/) and parsed the HTML coverage reports to extract line coverage information for tools comparison. In particular, we used this information to compute, for each tool pair, the number of statements covered by both tools and the number of statements covered by each of them separately. We added to each benchmark a broadcast receiver to save intermediate coverage results to disk (Dynodroid uses a similar strategy). This was necessary to collect coverage from the apps before they were restarted by the test input generation tools and also to track the progress of the tools at regular intervals. SwiftHand is an exception to this protocol. This tool, in fact, internally instruments the app under test to collect *branch* coverage and keep track of the app's lifecycle. This instrumentation, which is critical to the tool's functionality, conflicts with Emma's own instrumentation. We tried to resolve such conflict and even to map SwiftHand's to Emma's coverage, but we were not successful. We therefore do not compare the statement coverage information of SwiftHand with others. We nevertheless made the branch coverage information collected by SwiftHand on the benchmark apps available with our dataset.

To gather app failures, we collected the entire system log (also called `logcat`), from the emulator running the app under test. From these logs, we extracted failures that occurred while the app was being tested in a semi-automated fashion. Specifically, we wrote a script to find patterns of exceptions or errors in the log file and extract them along with their available stack traces. We manually analyzed them to ignore any failures not related to the app's execution (*e.g.,* failures of the tool themselves or initialization errors of other apps in the Android emulator). All unique instances of remaining failures were considered for our results.

TABLE II. LIST OF APPS USED IN OUR STUDY. (✓ INDICATES THAT THE APP WAS USED ORIGINALLY IN THE TOOL'S EVALUATION AND ⊗ INDICATES THAT THE TOOL CRASHED WHEN ATTEMPTING TO EXERCISE THE APP.)

Subject			Monkey	ACTEve	DynoDroid	A^{3}E	GuiRipper	PUMA	SwiftHand
Name	Ver.	Category							
Amazed	2.0.2	Casual	⊗		✓		⊗		
AnyCut	0.5	Productiv.			✓				
Divide&Conquer	1.4	Casual			✓		⊗		
LolcatBuilder	2	Entertain.			✓				
MunchLife	1.4.2	Entertain.			✓				
PasswordMakerPro	1.1.7	Tools			✓				
Photostream	1.1	Media			✓				
QuickSettings	1.9.9.3	Tools			✓				
RandomMusicPlay	1	Music		✓	✓				
SpriteText	-	Sample			✓				
SyncMyPix	0.15	Media			✓				
Triangle	-	Sample			✓				
A2DP Volume	2.8.11	Transport			✓				
aLogCat	2.6.1	Tools			✓				
AardDictionary	1.4.1	Reference			✓				
BaterryDog	0.1.1	Tools			✓				
FTP Server	2.2	Tools			✓				
Bites	1.3	Lifestyle			✓				
Battery Circle	1.81	Tools			✓				
Addi	1.98	Tools			✓				
Manpages	1.7	Tools			✓				
Alarm Clock	1.51	Productiv.			✓				
Auto Answer	1.5	Tools			✓				
HNDroid	0.2.1	News			✓				
Multi SMS	2.3	Comm.			✓				
World Clock	0.6	Tools			✓				⊗
Nectroid	1.2.4	Media			✓				
aCal	1.6	Productiv.			✓				
Jamendo	1.0.6	Music			✓				
AndroidomaticK.	1.0	Comm.		⊗	✓				
Yahtzee	1	Casual			✓				
aagtl	1.0.31	Tools			✓				
Mirrored	0.2.3	News			✓				
Dialer2	2.9	Productiv.			✓				
FileExplorer	1	Productiv.			✓				
Gestures	1	Sample			✓				
HotDeath	1.0.7	Card			✓				
ADSdroid	1.2	Reference	⊗		✓				
myLock	42	Tools			✓				
LockPatternGen.	2	Tools			✓				
aGrep	0.2.1	Tools	⊗	⊗	✓	⊗	⊗	⊗	
K-9Mail	3.512	Comm.			✓				
NetCounter	0.14.1	Tools			✓				⊗
Bomber	1	Casual			✓				
FrozenBubble	1.12	Puzzle	⊗		✓	⊗	⊗	⊗	
AnyMemo	8.3.1	Education		⊗	✓			⊗	✓ ⊗
Blokish	2	Puzzle			✓				
ZooBorns	1.4.4	Entertain.			✓				⊗
ImportContacts	1.1	Tools			✓				
Wikipedia	1.2.1	Reference		⊗	✓				
KeePassDroid	1.9.8	Tools			✓				
SoundBoard	1	Sample			✓				
CountdownTimer	1.1.0	Tools		✓					
Ringdroid	2.6	Media	⊗	✓ ⊗		⊗	⊗		
SpriteMethodTest	1.0	Sample		✓					
BookCatalogue	1.6	Tools					✓		
Translate	3.8	Productiv.		✓					
TomdroidNotes	2.0a	Social					✓		
Wordpress	0.5.0	Productiv.					✓		⊗
Mileage	3.1.1	Finance							✓
Sanity	2.11	Comm.							✓ ⊗
DalvikExplorer	3.4	Tools	⊗						✓
MiniNoteViewer	0.4	Productiv.							✓
MyExpenses	1.6.0	Finance							✓ ⊗
LearnMusicNotes	1.2	Puzzle							✓
TippyTipper	1.1.3	Finance							✓
WeightChart	1.0.4	Health							✓ ⊗
WhoHasMyStuff	1.0.7	Productiv.							✓

TABLE III. EASE OF USE AND COMPATIBILITY OF EACH TOOL WITH THE MOST COMMON ANDROID FRAMEWORK VERSIONS.

Name	Ease Use	Compatibility
Monkey [24]	NO_EFFORT	any
Dynodroid [18]	NO_EFFORT	v.2.3
GUIRipper [1]	MAJOR_EFFORT	any
A^{3}E-Depth-first [6]	LITTLE_EFFORT	any
SwiftHand [8]	MAJOR_EFFORT	v.4.1+
PUMA [13]	LITTLE_EFFORT	v.4.3+
ACTEve [3]	MAJOR_EFFORT	v.2.3

C1: Ease of Use

Tools should ideally work out of the box, and should not require extra effort of the user in terms of configuration. Table III reports whether the tool worked out of the box (NO_EFFORT), whether it required some effort (LITTLE_EFFORT), either to properly configure it or to fix minor issues, or whether it required a major effort (MAJOR_EFFORT) to make it work. This is based on our judgment and experience, and the evaluation does not have the value of a user study, mainly because most of these tools are just early prototypes. In future we would like to assess the ease of use of each tool through a proper user study, but as of now, we simply report our experience with installing each tool, and we describe the required fixes to make each of them work. Some of the changes were required to make the tools run on our infrastructure.

Monkey: We used the vanilla Monkey from the Android distribution for our experimentation. The tool was configured to ignore any crash, system timeout, and security exceptions during the experiment and to continue till the experiment timeout was reached. In addition, we configured it to wait 200 milliseconds between actions, as this same delay value was also used in other tools. Configuring Monkey for our infrastructure required no extra effort.

ACTEve: We consulted the authors to apply minor fixes to the instrumentation component of ACTEve, which instruments both the Android SDK and the app under test. While generating tests ACTEve often restarts the app. To ensure that we did not lose coverage information, we modified ACTEve to save the intermediate coverage before app restarts.

GUIRipper: As discussed in Section III-B, at the time of this study GUIRipper was available only as a Windows binary distribution, so we had to put some effort into porting it to our Linux based infrastructure. We configured the tool to use its systematic ripping strategy instead of the default random one. Because GUIRipper often restarts the Android emulator from a snapshot to return to the initial state of an app, we modified the tool to save intermediate coverage before such restarts.

Dynodroid: We obtained a running version of Dynodroid from the virtual machine provided on the tool's page. Dynodroid is tightly coupled with the Android emulator, for which the tool includes an instrumented system image. In addition, the tool performs an extensive setup of the device after boot before starting the exploration. We configured our timer to start counting at the start of the actual exploration to account for this one-time setup time.

A^{3}E: We updated A^{3}E's dependencies to make it work with the latest version of the Android SDK. The public release only supports the depth-first exploration strategy for systematic testing, so this is what we used in our experiments. In addition, we modified the tool to report verbose results for Android commands invoked, and to not shutdown the emulator after input generation to allow us to collect reports from it.

SwiftHand: The SwiftHand tool consists of two components: front-end, which performs bytecode instrumentation of the app under test, and back-end, which performs test input generation. We fixed the dependencies of the front-end tool by obtaining an older version of the `asmdex` library (http://asm.ow2.org/asmdex-index.html) and wrote a wrapper shell script to connect the two components in our infrastructure.

PUMA: To get PUMA running, we applied a minor patch to use an alternate API for taking device screenshots. We also altered the different timeouts in the tool to match our experimental settings.

In summary, we found Monkey and Dynodroid straightforward to configure and use, while GUIRipper, ACTEve and A^{3}E required major efforts for their configuration.

C2: Android Framework Compatibility

Android developers have to constantly deal with the fragmentation problem, that is, their apps must run on devices that have different hardware characteristics and use different versions of the Android framework. It is thus desirable, for a test input generator tool, to be compatible with multiple releases of the Android framework. Therefore, we ran each tool on three popular Android framework releases, as described in Section IV-B, and assessed whether it could work correctly.

Table III reports the results of this study. The table shows that 4 out of 7 tools do not offer this kind of compatibility. Some tools (PUMA and SwiftHand) are compatible only with the most recent releases of the Android Framework, while others (ACTEve and Dynodroid) are bound to a very old one. ACTEve and Dynodroid may be compatible with other versions of the Android framework, but this would require to instrument these versions first. SwiftHand and PUMA, conversely, are not compatible with older versions of Android because they use features of the underlying framework that were not available in previous releases.

C3: Code Coverage Achieved

Test input generation tools for Android implement different strategies to explore as much behavior as possible of the app under test. Section III presented an overview of the three main strategies used by these tools: random, model-based, and systematic. Although some of these tools have been already evaluated according to how much code coverage they can achieve, it is still unclear whether there is any strategy that is better than others in practice. Previous evaluations were either incomplete because they did not include comparisons with other tools, or somehow biased (in our opinion). Since we believe that the most critical resource, when generating inputs, is *time*, tools should evaluate how much coverage they can achieve within a certain time limit. Tools such as Dynodroid and EvoDroid, however, have been compared to Monkey by comparing the coverage achieved given the same *number of generated events*. In addition, most studies did not account for the non-determinisms of the Android environment and ran the tools only once. To address these shortcomings, in our experiment, we ran each tool on each app of Table II for the same amount of time and for 10 times, as described in Section IV-B.

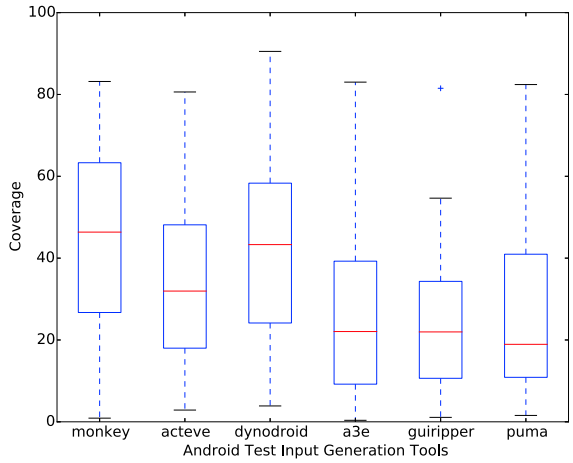

Fig. 1. Variance of coverage achieved across apps and over 10 runs.

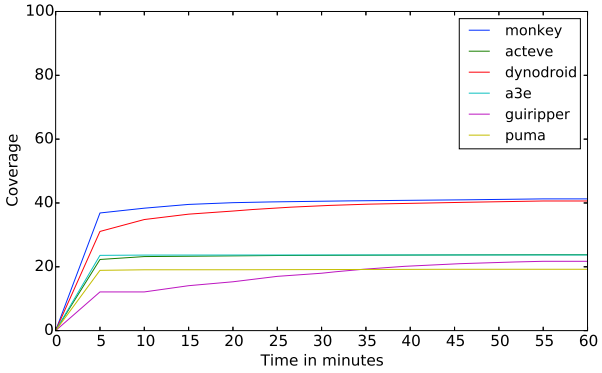

Fig. 2. Progressive coverage.

Figure 1 reports the variance of the mean coverage of 10 runs that each tool achieved on the considered benchmarks. We can see that, on average, Dynodroid and Monkey performed better than other tools, followed by ACTEve. The other three tools (*i.e.,* A^{3}E, GUIRipper and PUMA) achieved a fairly low coverage level. Despite this, even those tools that on average achieved low coverage could reach very high coverage (approximately 80%) for a few apps. We manually investigated these apps and found that they are, as expected, the simplest ones. The two outliers, for which every tool achieved very high coverage, are DivideAndConquer and RandomMusicPlayer. The first one is a game that accepts only touches and swipes as events, which can be provided without much logic in order to proceed with the game. Also for RandomMusicPlayer, the possible user actions are quite limited, as there is only one activity with four buttons. Similarly, there are some apps for which every tool, even the ones that performed best, achieved very low coverage (*i.e.,* lower than 5%). Two of these apps, K9mail (a mail client) and PasswordMakerPro (an app to generate and store authentication data), highly depend on external factors, such as the availability of a valid account. Such inputs are nearly impossible to generate automatically, and therefore every tool stalls at the beginning of the exploration. A few tools do provide an option to manually interact with an app initially, and then use the tool to perform subsequent test input generation. However, we did not use this feature for scalability reasons and concerns of giving such tools an unfair advantage.

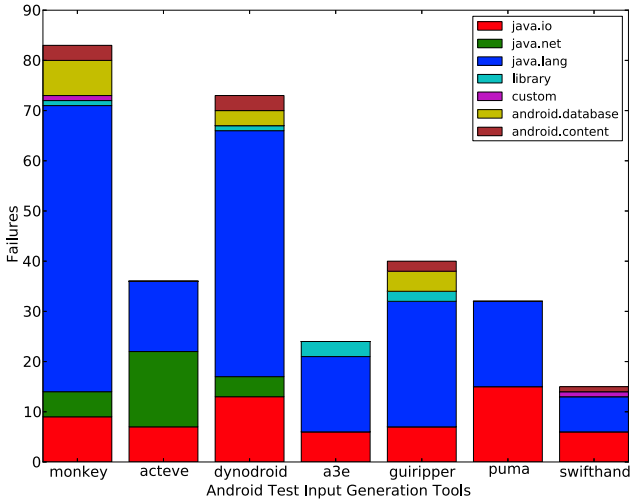

Fig. 3. Distribution of the failures triggered by the tools.

Figure 2 reports the progressive coverage of each tool over the time threshold we used (*i.e.,* 60 minutes). The plot reports the mean coverage achieved across all apps over 10 runs. This plot shows that all tools hit their maximum coverage within a few minutes (between 5 and 10), with the only exception of GUIRipper. The likely reason for this difference is that GUIRipper frequently restarts the exploration from its initial state, and this operation takes time. (This is in fact the main problem that SwiftHand addresses by implementing an exploration strategy that limits the number of restarts.)

C4: Fault Detection Ability

The main goal of test input generation tools is to expose faults in the app under test. Therefore, beside code coverage, we checked how many failures each tool could reveal within a one-hour per app time budget. None of the Android tools can identify failures other than runtime exceptions, although there is some promising work that goes in that direction [35].

Figure 3 reports the results of this study. The y axis represents the number of *cumulative unique failures* across the 10 runs across all apps. We consider a failure unique when its stack trace differs from the stack trace of other failures. The plot also reports some high level statistics about the most frequent failures (we report the package name of the corresponding runtime exception). Only a few of these failures involved custom exceptions (*i.e.,* exceptions that are declared in the app under test). The vast majority of them resulted in standard Java exceptions, among which the most frequent ones are null pointer exceptions.

Because SwiftHand is the tool with the worst performance in this part of the evaluation, we looked into its results in more detail to understand the reasons behind that. We found that SwiftHand crashes on many of our benchmarks, which prevents it from producing useful results in these cases. Further analysis revealed that the crashes are most likely due to SwiftHand's use of `asmdex` to instrument the apps it is testing. The `asmdex` framework is in fact not well maintained and notoriously crash-prone.

Figure 4 reports a pairwise comparison of each tool according to coverage (upper part of the figure—boxes with

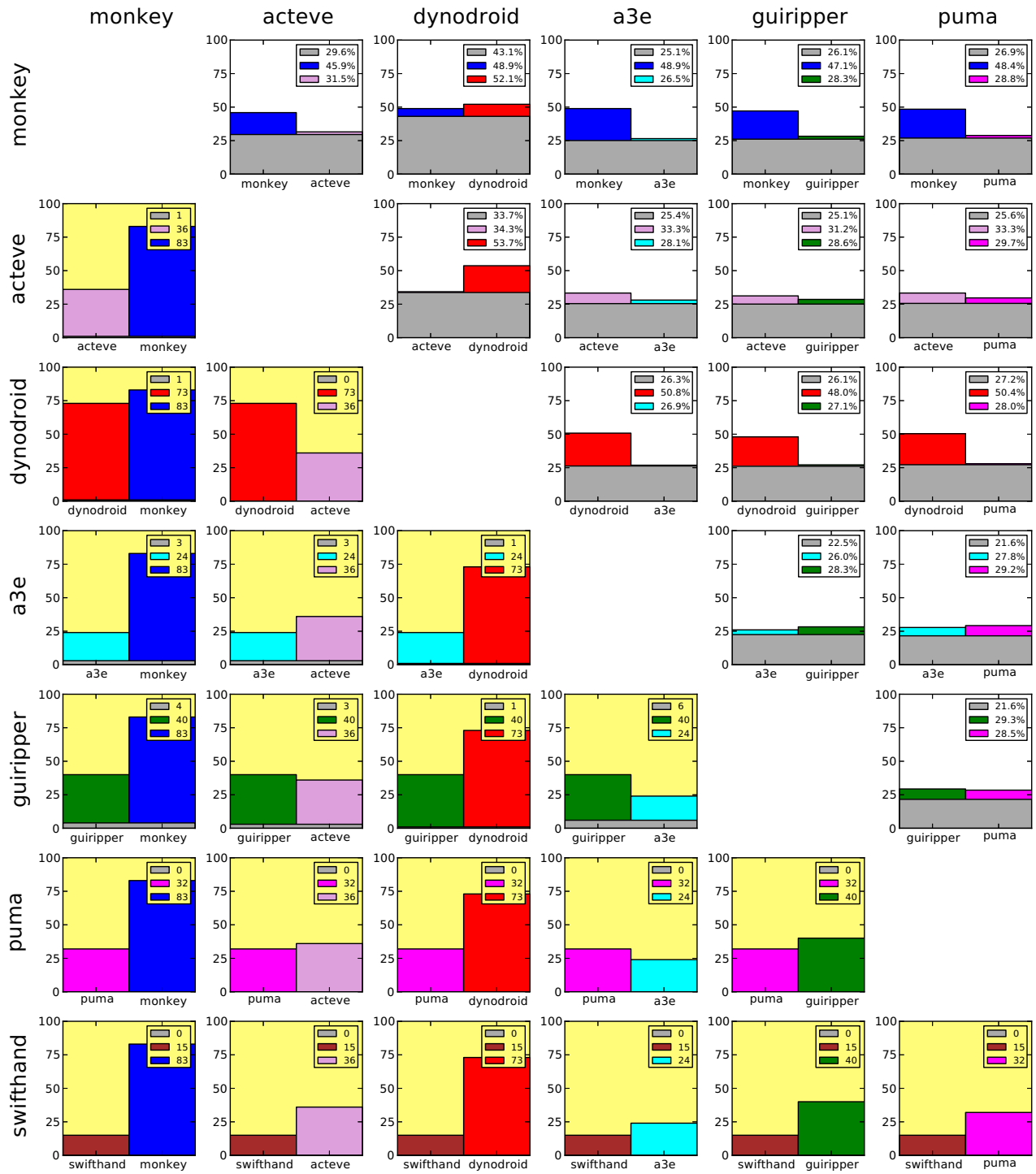

Fig. 4. Pairwise comparison of tools in terms of coverage and failures triggered. The plots on the top right section show percent statement coverage of the tools, whereas the ones in the bottom left section show absolute number of failures triggered. The gray bars in all plots show commonalities between the results of the corresponding two tools.

white background) and fault detection ability (lower part of the figure—boxes with yellow background). In the figure, we compare the coverage of the best out of the 10 runs of each tool, whereas the number of failures reported are the cumulative failures with the same stack trace across the 10 runs. The figure shows both which lines are covered (and which failures are reported) by both tools (in grey) and which lines are covered by only one of the two tools. These results show that tools do

not complement each other in terms of code coverage, whereas they do in terms of fault detection. In fact, while for coverage the common parts are significant, it is almost the opposite in terms of failures.

V. Discussion and Future Research Directions

The experiments presented in Section IV produced a somehow unexpected result, as they show that the random exploration strategies implemented by Monkey and Dynodroid achieved higher coverage than more sophisticated strategies in other tools. These may indicate that Android apps are different from Java stand-alone application, where random strategies tend to be inefficient compared to more systematic strategies (*e.g.,* [12], [21], [22]). In fact, our results seem to indicate that (1) app behavior can be suitably exercised by generating only UI events, and (2) a random approach may be effective enough in this context.

Considering the four criteria considered in the study, Monkey would be a clear winner among the test input generation tools considered: it achieved the best coverage on average, reported the largest number of failures, was easy to use, and worked on every platform. This does not mean, however, that the other tools should not be considered—every tool showed strengths that, if properly leveraged and combined, may lead to significant overall improvements. In this spirit, we discuss some of the features provided by these tools that seemed to make a difference in their performance and should therefore be considered by other tools as well.

Generating system events. Dynodroid and ACTEve can generate system events beside standard UI events. Even if the behavior of an app may depend only partially on system events, generating them can reveal failures that would be hard to uncover otherwise.

Minimizing restarts. Progressive coverage shows that tools frequently restart their exploration from scratch need more time to achieve their maximum coverage. The search algorithm that Swifthand implements aims to minimize such restarts and allows it to achieve higher coverage in less time.

Allowing for manually provided inputs. Specific behaviors can sometimes only be explored by specific inputs (*e.g.,* logins and passwords), which may be hard to generate randomly or by means of systematic techniques. Some tools, such as Dynodroid and GUIRipper, let users manually provide values that the tool can later use during the analysis.

Considering multiple starting states. The behavior of many apps depends on the underlying content providers. An email client, for instance, would show an empty inbox, unless the content provider contains some messages. GUIRipper starts exploring the app from different starting states (*e.g.,* when the content provider is empty and when it has some entries). Even if this has to be done manually by the user, by properly creating snapshots of the app, it allows a tool to potentially explore behavior that would be hard to explore otherwise.

Avoiding side effects among different runs. Although in our experiments we used a fresh emulator instance between runs, we realized that some tools, such as Dynodroid and A^{3}E, provided the ability to (partially) clean up the environment by uninstalling the app under test and deleting its data. Input generation tools should reuse the environment for efficiency reasons but should do it in a way that does not introduce side effects.

In our study, we also identified limitations that can significantly affect the effectiveness of a tool. We report these limitations, together with desirable and missing features, such that they could be the focus of future research in this area:

Reproducibility. None of the tools studied allowed for easily reproducing the failures they identified. Although they reported uncaught runtime exceptions on their logfiles, they did not generate test cases that could be later rerun. We believe that this is an essential feature that every input generation tool should provide.

Mocking. Most apps for which tools had low coverage highly depended on environment elements, such as content providers. GUIRipper alleviates this problem by letting users prepare different snapshots of the app. We believe that working on a proper, more general mocking infrastructure for Android apps would be a significant contribution and could lead to drastic code coverage improvements.

Sandboxing. Input generation tools should also provide proper sandboxing, that is, they should block potentially harmful operations (*e.g.,* sending messages or deleting files). None of the tools considered took this problem into account.

Addressing the fragmentation problem. While our evaluation showed that some tools can run on multiple versions of the Android framework, none of them is specifically designed for cross-device testing. While this is a somehow specific testing problem, we believe that testing tools for Android should also consider this aspect, as fragmentation is arguably one of the major problems that Android developers have to face.

VI. Conclusion

In this paper, we presented a comparative study of the main existing test input generation tools (and corresponding techniques) for Android. We evaluated these tools according to four criteria: ease of use, Android framework compatibility, code coverage achieved, and fault detection ability. Based on the results of this comparison, we identified and discussed strengths and weaknesses of the different techniques and highlighted potential venues for future research in this area. To facilitate reproduction of our results and facilitate empirical studies of this kind, all of our experimental infrastructure and data are publicly available at http://www.cc.gatech.edu/~orso/software/androtest.

In future work, we will extend our empirical comparison of input generation tools for Android apps by (1) considering additional benchmarks and (2) possibly including to the set of tools considered also the intent fuzzers that we excluded from the current study.

Acknowledgments

We thank the authors of the tools we studied for making their tools available and helping with the tools setup. This work was supported in part by NSF grants CCF-1161821 and CCF-1320783 and by funding from Google, IBM Research, and Microsoft Research to Georgia Tech.

REFERENCES

[1] 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*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR – a tool for automated model-based testing of mobile apps," *IEEE Software*, vol. PP, no. 99, pp. NN–NN, 2014.

[3] 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*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://www.bodden.de/pubs/far+14flowdroid.pdf

[5] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15. ACM, May 2015, pp. 426–436.

[6] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509549

[7] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, ser. SOAP '12. New York, NY, USA: ACM, 2012, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259056

[8] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 623–640. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509552

[9] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "WebMate: Generating Test Cases for Web 2.0," in *Software Quality. Increasing Value in Software and Systems Development*. Springer, 2013, pp. 55–69.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[11] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE '14. New York, NY, USA: ACM, June 2014, pp. 1025–1035. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568276

[12] F. Gross, G. Fraser, and A. Zeller, "EXSYST: Search-based GUI Testing," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1423–1426. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337435

[13] 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*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 204–217. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594390

[14] C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982612

[15] M. Kechagia, D. Mitropoulos, and D. Spinellis, "Charting the API minefield using software telemetry data," *Empirical Software Engineering*, pp. 1–46, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-014-9343-7

[16] Y. Liu, C. Xu, and S. Cheung, "Verifying android applications using java pathfinder," The Hong Kong University of Science and Technology, Tech. Rep., 2012.

[17] H. Lockheimer, "Google bouncer," http://googlemobile.blogspot.com.es/2012/02/android-and-security.html.

[18] 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*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[19] 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*, ser. FSE 2014. New York, NY, USA: ACM, 2014.

[20] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '12, 2012, pp. 1–12.

[21] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1109/ICST.2012.88

[22] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–. [Online]. Available: http://dl.acm.org/citation.cfm?id=950792.951350

[23] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[24] "The Monkey UI android testing tool," http://developer.android.com/tools/help/monkey.html.

[25] "Intent fuzzer," 2009, http://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx.

[26] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android Applications to Java Bytecode," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393600

[27] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-PERT: Accurate Identification of Cross-browser Issues in Web Applications," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 702–711. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486881

[28] 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)*, ser. WODA+PERTEA 2014. New York, NY, USA: ACM, 2014, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/2632168.2632169

[29] "Smali/baksmali, an assembler/disassembler for the dex format used by Dalvik," https://code.google.com/p/smali.

[30] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying Android Applications Using Java PathFinder," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012. [Online]. Available: http://doi.acm.org/10.1145/2382756.2382797

[31] ——, "Execution and Property Specifications for JPF-android," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–5, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2557833.2560576

[32] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "ProfileDroid: Multi-layer Profiling of Android Applications," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, ser. Mobicom '12. New York, NY, USA: ACM, 2012, pp. 137–148. [Online]. Available: http://doi.acm.org/10.1145/2348543.2348563

[33] 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*, ser. FASE'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 250–265. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37057-1_19

[34] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, ser. MoMM '13. New York, NY, USA: ACM, 2013, pp. 68:68–68:74. [Online]. Available: http://doi.acm.org/10.1145/2536853.2536881

[35] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 183–192. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.31