symmetry

MDPI

Article

An Empirical Analysis of Test Input Generation Tools for Android Apps through a Sequence of Events

Husam N. Yasin [1,*], Siti Hafizah Ab Hamid [1,*], Raja Jamilah Raja Yusof [1] and Muzaffar Hamzah [2,*]

[1] Department of Software Engineering, Faculty of Computer Science and Information Technology, University Malaya, Kuala Lumpur 50603, Malaysia; rjry@um.edu.my
[2] Faculty of Computing and Informatics, Universiti Malaysia Sabah, Sabah 88400, Malaysia
* Correspondence: husam_yas@siswa.um.edu.my (H.N.Y.); sitihafizah@um.edu.my (S.H.A.H.); muzaffar@ums.edu.my (M.H.)

Received: 3 November 2020; Accepted: 16 November 2020; Published: 18 November 2020



Abstract: Graphical User Interface (GUI) testing of Android apps has gained considerable interest from the industries and research community due to its excellent capability to verify the operational requirements of GUI components. To date, most of the existing GUI testing tools for Android apps are capable of generating test inputs by using different approaches and improve the Android apps' code coverage and fault detection performance. Many previous studies have evaluated the code coverage and crash detection performances of GUI testing tools in the literature. However, very few studies have investigated the effectiveness of the test input generation tools, especially in the events sequence length of the overall test coverage and crash detection. The event sequence length generally shows the number of steps required by the test input generation tools to detect a crash. It is critical to highlight its effectiveness due to its significant effects on time, testing effort, and computational cost. Thus, this study evaluated the effectiveness of six test input generation tools for Android apps that support the system events generation on 50 Android apps. The generation tools were evaluated and compared based on the activity coverage, method coverage, and capability in detecting crashes. Through a critical analysis of the results, this study identifies the diversity and similarity of test input generation tools for Android apps to provide a clear picture of the current state of the art. The results revealed that a long events sequence performed better than a shorter events sequence. However, a long events sequence led to a minor positive effect on the coverage and crash detection. Moreover, the study showed that the tools achieved less than 40% of the method coverage and 67% of the activity coverage.

Keywords: android apps; empirical study; GUI testing; test input generation

1. Introduction

Android applications (or mobile apps) play a vital role in our daily lives. Previous research work [1] has shown that the number of Android apps downloaded has increased drastically over the years. With over 85% shares of the global market, Android is one of the most popular mobile operating systems in more than 2 billion active devices monthly around the world [2]. Google Play Store is the official market of Android apps with more than 3.3 million apps. Unfortunately, in December 2019, it was reported by AppBrain that about 17% of the Android apps were low-quality apps [3]. Furthermore, it was also reported that approximately 53% of the smartphone users did not use the app once the app crashed [4]. Mobile app crash is evitable and avoidable through intensive and extensive mobile app testing. Mobile apps can be tested with a graphical user interface (GUI) tool to verify the mobile app's functionality, usability, and consistency before they are released to the market [5–7]. To start a mobile

app's testing, test cases will be generated with a series of events sequenced from the GUI components. In the mobile apps, the test input (or test data) can be either from user interaction or system interaction (e.g., SMS notification). The development of GUI test cases usually takes a great deal of times and effort due to their non-trivial structures and the highly interactive nature of GUIs. Android apps [8,9] usually possess numerous states and transitions, which can lead to an arduous testing process and poor testing performance. For the past decade, Android test input generation tools have been developed to automate user interactions and allow system interactions as inputs [10–15]. However, the existing tools can only determine the crashes caused by user inputs [8,16–18]. Previous empirical studies have evaluated the performance of several test input generation tools in terms of test coverage and crash detection on open-source apps [8] and industrial apps [19]. Choudhary et al. [8] and Wang et al. [19] have demonstrated that Android Monkey [20] was the best test input generation tool for Android apps. Android Monkey was user-friendly and able to generate a vast number of random events for stress testing. However, Android Monkey covered less than 50% of the code coverage in the open-source apps [8] and industrial apps [19]. Therefore, it is no surprise that Android Monkey is yet to become a standard testing tool [21]. Nevertheless, to the best of our knowledge, no studies have investigated the effectiveness of events sequence length on code coverage and crash detection. Thus, it is critical to investigate the effectiveness of events sequence length due to its significant effects on time, testing effort, and computational cost.

In this paper, an experimental analysis was performed to investigate the impact of events sequence length on the code coverage and crashes detection. The main contributions of this paper are as follows:

- An extensive comparative study of test input generation tools performed on 50 Android apps using six Android test input generation tools.
- An analysis of the strengths and weaknesses of the six Android test input generation tools.

The rest of this paper is divided as follows. Section 2 presented the background of Android apps GUI and Section 3 discussed the test input generation tools for Android apps. For Section 4, the design of the case study is elucidated. Next, Section 5 illustrated the execution steps of the case study while Section 6 analyzed and discussed the findings. In Section 7, future research directions that should be considered by other GUI testing tools are highlighted. Finally, Section 8 discussed the possible threats to the validity of the results before concluding the findings in Section 9.

2. GUI of Android Apps

Activity is the main interface for user interaction and each activity represents a group of layouts. For example, a linear layout organizes the screen items horizontally or vertically. The interface has GUI elements, which are also known as widgets or controls. These widgets include buttons, text boxes, search bars, switches, and number pickers which allow the users to interact with the apps. As a whole, it can be categorized into four attributes: type (e.g., class), appearance (e.g., text), functionalities (e.g., clickable and scrollable), and the designated order of the sibling widgets (i.e., index). These widgets are handled as the task stacks in the system. The layouts and widgets are described in the manifest file of Android apps, where each layout and widget have a unique identifier. The manifest file of an Android app is an important XML (eXtensible Markup Language) file which is stored in the root directory of the app's source as AndroidManifest.xml. The binary manifest file has the essential Android system details of the device such as the package name, App ID, the minimum level of API (Application Programming Interface) needed, the list of permissions required, and the hardware specifications.

For Android apps, activity is the primary target of testing tools as the users navigate through the screen. The four methods used to describe a full lifecycle of activity include (1) created, (2) paused, (3) resumed, and (4) destroyed. All these four methods can be linked and disabled when the activity status changed. An activity can interrupt and change its lifecycle status unexpectedly with the user interactions or environment changes, e.g., from foreground to background by phone calls. The lifecycle of an activity is tightly coupled with the Android framework, which is also managed by an essential

service or better known as the activity manager [22]. The activity manager provides numerous methods to start an activity. For example, the initiation of activity requires one to invoke a method such as startActivity to notify the activity manager of the request. Successively, a method such as startActivity requires a parameter and intent that contains the URI (Uniform Resource Identifier) of the activity. Figure 1 shows the lifecycle of the Android app activity.

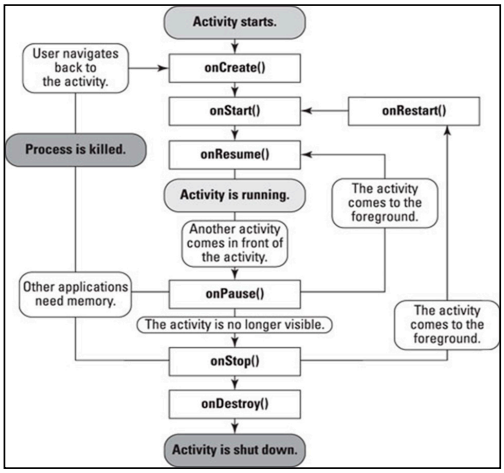


Figure 1. Android application activity lifecycle.

3. Android Test Input Generation Tools

Test input generation tools identify GUI elements and produce relevant inputs to exercise the behavior of an app by using computer science algorithms [23]. Android apps are tested with sequences of events automatically generated to simulate the user's interaction with the GUI. For example, an interaction usually involves clicking, scrolling, or typing texts into a GUI element such as a button, image, or text block. Furthermore, Android apps can sense and respond to a large number of inputs from the system interactions [24]. Interaction with system events includes receiving SMS notifications, app notifications, or phone calls. These experiences are some of the events that need to be addressed in the testing of Android apps, as they increase the complexity of apps testing effectively [25]. In GUI testing, a sequence of events is an integral part of a GUI test case. A GUI test case consists of (1) a sequence of events (e.g., button clicks), (2) potential input values, and (3) expected results from the test case. The generation of test cases is one of the most demanding tasks because it has substantial impacts on the effectiveness and efficiency of the entire testing process [26].

The main goal of Android test input generation tools is to identify the faults of the App Under Test (AUT). The tools produce a sequence of events that represent the possible cases to exercise the AUT's functionality while covering as much source code as possible. They can generate inputs with different exploration strategies. Exploration can either be guided by a model of the app, which constricts static, dynamic, or exploit techniques that aim to achieve more code coverage. In another perspective, test input generation tools can generate events that consider Android apps as a black-box or grey-box. These tools extract high-level properties of the app including the list of activities and UI elements of each activity, in order to generate events that will likely expose new behavior. App developers are the main stakeholders of these tools. By using these generation tools, the developers can test the app automatically and fix the potential issues before using it. In the next section, each tool is grouped and categorized based on their respective approaches.

In Sections 3.2 and 3.3, the black and grey-boxes of the Android test input generation tools are explained in detail. Table 1 shows an overview of the existing Android test input generation tools that support the system events generation from the literature.

Table 1. Overview of Android test input generation tools understudy.

No	Tool	Approach	Exploration Strategy	Events	Method of System Events Identification	Crash Report	Replay Scripts	Testing Environment	Basis	Code Availability
1	Humanoid	Black box	Deep Q Network	User, System	Dynamic Analysis (dumpsys)	Log	Yes	Real Device, Emulator	Droidbot	Yes
2	AndroFrame	Black box	QLearning-Based	User, System	Dynamic Analysis (App's Permission)	-	Yes	Real Device, Emulator	-	No
3	DroidBot	Black box	Model-based	User, System	Dynamic Analysis (dumpsys)	Log	Yes	Real Device, Emulator	-	Yes
4	SmartMonkey	Black box	Random based	User, System	Dynamic Analysis (App's Permission)	-	-	-	Monkey	No
5	Stoat	Grey box	Model-based	User, System	Static analysis	Text	Yes	Real Device, Emulator	A3E	Yes
6	Sapienz	Grey box	Search-based/Random	User, System	Static analysis	Text	Yes	Emulator	Monkey	Yes
7	Crashscope	Grey box	Systematic	User, System	Dynamic Analysis (App's Permission)	Text, Image	Yes	Real Device, Emulator	-	No
8	Dynodroid	Black box	Guided/Random	User, System	Dynamic Analysis (App's Permission)	Log	No	Emulator	Monkey	Yes
9	ExtendedRipper	Black box	Model-based	User, System	Dynamic Analysis (Events Pattern)	Log	No	Emulator	GUI Ripper	No
10	A3E-Targeted	Grey box	Systematic	User, System	Static analysis	-	No	Real Device, Emulator	Troyd	No
11	Monkey	Black box	Random based	User, System	Dynamic Analysis (dumpsys)	Log	No	Real Device, Emulator	-	Yes

3.1. Features of Android Test Input Generation Tools

Nine features were used to describe the Android test input generation tools.

3.1.1. Approach

Three main approaches of Android test input generation tools were applied in the literature: (1) white-box approach used to validate the Android apps based on the source code of the apps, (2) black-box approach used to test and check the validity of the Android apps' GUI based on the system requirements, and (3) grey-box approach (a combination of white-box and black-box approaches) used to analyze the suitability of usage and structure based on the architecture or pattern/design.

3.1.2. Exploration Strategy

There are five exploration strategies used in the test input generation tools: (1) random-based, (2) model-based, (3) systematic, (4) search-based, and (5) reinforcement learning.

(1) Random-based strategy is one of the most popular techniques used to detect faults in the apps. It generates a randomized series of infeasible input events that usually fail to explore all the app functionalities [27]. This technique is suitable to generate events that allow stress testing. However, this technique tends to repeat its actions, which leads to low test coverage and high execution time. Furthermore, this technique requires the developers to specify the timeout manually.

(2) Model-based strategy uses a directed graph-based model to correlate the relationship of the user interaction and the GUI of the apps. It uses the depth first strategy (DFS) algorithm in traversing through the nodes (the GUI widgets) and edges (the user interaction) [28]. The model is designed either manually or automatically by adopting the specifications of the AUT such as code, XML configuration files, and direct interactions with apps. A model-based exploration technique produces better code coverage than random testing due to a smaller number of redundant inputs. However, this technique has the drawback of generating inaccurate modeling. More specifically, the dynamic behaviors of the GUIs tend to produce erroneous model or state explosion issues due to the non-deterministic changes in the GUIs. Thus, the model-based approach will overlook the changes and search for trivial events before proceeding with the discovery differently. Explicitly, a GUI model with a limited range of possible behavioral spaces will affect the effectiveness of tests adversely.

(3) Systematic exploration uses more sophisticated techniques such as symbolic execution and evolutionary algorithms. It reveals the codes by random-based and model-based strategies. These techniques can explore some of the app's behavior with specific inputs. The main benefit of this technique is it can leverage the source code to reveal previously uncovered app behavior.

(4) Search-based strategy uses meta-heuristic search algorithms [29] such as genetic algorithms (GA) for testing. This technique was adopted in Sapienz [13], who used GA to optimize randomly generated inputs to achieve multi-objectives.

(5) Reinforcement learning (RL) is a machine learning approach that focuses on behavioral psychology, which interacts directly with the component's environment [30]. RL techniques include Q-learning, Deep Q Network (DQN), Actor-critic, and State-Action-Reward-State-Action (SARSA). In the context of Android apps testing, AUT is the environment, and the state is a set of actions available on the activity of AUT. GUI actions are a set of actions available in the current state of the environment, and the testing tool is the agent. Initially, the testing tool has limited knowledge about the AUT. It is updated throughout as the tool generates and executes the test input. The tool uses the knowledge gained to make effective decisions for future actions.

3.1.3. Events

In test input generation, events refer to the data used to execute the test cases of events. An Android app receives two types of events [31]: user events and system events. User events are the interactions between the user and the app through the interface inputs such as touch, text, and scrolls.

System events are messaging systems inside the apps and outside of the regular user flow such as receiving SMS notification, apps notification, and phone calls. For example, Android apps enable us to send or receive broadcast messages from the Android system and other Android apps. The broadcast messages are usually enclosed in an intent object. The action string of the intent object will identify and store the event data.

3.1.4. Methods of System Events Identification

System events are extracted from the AUT using two approaches. (1) Static analysis analyzes the Android app's events handler statically and detects the system events based on the source code of the AUT without executing the app. The static analysis reviews and detects faults which could potentially serve as a failure cause in the source code. However, it is only limited in the design and implementation phases and offers support for the Dalvik bytecode analysis. (2) Dynamic analysis explores the GUI of AUT to extract all possible event sequences. It executes the AUT by using different techniques to observe the external behavior of AUT. Dumpsys is a tool that runs on Android devices, which provides information about system services. To obtain a diagnostic output from all system services running on a connected device, a command-line can be called using the Android Debug Bridge (ADB) through a dynamic analysis approach. Event patterns are a sequence of system events used to exercise the app. The sequence is defined manually with proper regular expressions like optional, mandatory, and iterative events. Besides, the Android Operating System (OS) uses a permission-based approach to control the behavior of Android apps and the accessibility of sensitive data (e.g., photo gallery, calls log, contacts) and components (e.g., GPS, Camera) on the Android devices. The permission required within the Android apps is declared in the Android manifest file and the dynamic approach analyzes the Android manifest file to identify the system events.

3.1.5. Crash Report

An Android app crashes when an unexpected exit is induced and an unhandled exception occurs [32]. Test input generation tools detect the crashes and provide the app developers with a comprehensive crash report. The crash report contains a captured stack trace which indicates the location of the crash from the source code of the AUT. Moreover, screenshots, natural language reproduction steps, and replay-able scripts are provided as well. The report is presented in the form of a log, image, or text.

3.1.6. Replay Scripts

The tools generate reproducible test cases, which can be used to replay the crashes on the target device(s). Reproducibility of test cases is essential as it allows the analysis of different app behaviors and improves the quality of the apps. The tools detect faults, locate triggers, and conduct regression tests after the faults have been fixed. The same reproducible test case can be executed on different device configurations to detect compatibility issues. When a fault is detected, the fault is usually reproduced at least twice before the developers fix the error.

3.1.7. Testing Environment

The testing environment is a platform that supports the execution of hardware and software tests. The test environment is typically designed based on the specifications of the AUT. There are two main options in conducting GUI tests on Android apps: (1) Real device, and (2) Emulator. A real device refers to a mobile device (phone or tablet). The real device is the ultimate way to understand the app users' experience. Real devices produce real results and live network performance defects. On the contrary, an emulator is a virtual mobile device that runs on a computer to simulate some features of the hardware and software in a real device. It can be used to test the app by using a massive device fragmentation of the Android domain. Developers often use on-screen Android emulators to test Android apps in a digital environment. An emulator is part of the Android software development kit

(SDK). However, the emulator is unable to simulate events such as battery issues, network connectivity, and gestures.

3.1.8. Basis

The basis is the underlying tool to design and build new tools. For example, Smart-Monkey [33] was built on top of Android Monkey [20] and such a configuration was more efficient in detecting GUI bugs and valid events. As Android Monkey is not limited to Smart-Monkey, several tools have advanced versions of Android Monkey, such as Sapienz [13] and Dynodroid [10]. GUIRipper [34] served as a base tool for ExtendedRipper [35] which was used to restart the exploration from the initial state. It also generates system events and covers a wider code coverage than its bases tool. DroidBot [36], an open-source testing tool used to generate events in the GUI and system. In addition, Humanoid [37] is an upgraded version of DroidBot. It predicts that human users are more likely to interact with UI elements. Besides, Stoat [12] is an upgraded version of the A3E [38] and is used due to its unavailability to the public repository. However, Stoat has an enhanced UI exploration strategy and static analysis.

3.1.9. Code Availability

In this context, availability refers to the availability of the source code to be accessed by the public. Despite Sapienz source code is available on GitHub, it is outdated and unsupported. While other tools such as Stoat, DroidBot, Humanoid, and Dynodroid source codes are easily accessible by the public and constantly updated on GitHub. On the contrary, Android Monkey is available on the Android SDK. For Crashscope, the source code is unavailable, but the tool is user-friendly on the internet. ExtendedRipper only supports Windows operating systems. Furthermore, the source codes of A3E-Targeted, Androframe, and Smart-Monkey are also unavailable on the internet.

3.2. Black Box Android Test Input Generation Tools

Humanoid [37] was implemented along with DroidBot [36] which was developed to learn how users interact with Android apps. Humanoid uses a GUI model to comprehend and analyze the behavior of AUT. Nonetheless, Humanoid prioritizes human interacted UI elements. Humanoid operates in two phases; (1) offline learning phase which is a deep neural network model used to master the relationship between GUI contexts and user-performed interactions, and (2) online testing phase where Humanoid developed a GUI model for the AUT. In the second phase, it uses the GUI model and the interaction model to determine the type of test input to send. The GUI model directs Humanoid on the navigation of explored UI states, while the interaction model guides the discovery of the new UI states.

Androframe [39] generates test cases based on a Q-learning approach. Instead of using a random exploration approach, the GUI is explored based on a pre-approximated probability distribution that satisfied a test objective. Before it is used in the next action, it creates a Q-matrix with the probabilities of achieving the test objective. However, Androframe has inconsistent activity coverage and only works with single-objective fitness functions, where each run has only one objective to increase the activity coverage or search crashes.

DroidBot [36] is an open-source testing tool that utilizes a model-based exploration strategy to generate user events and system events under a black-box approach. Furthermore, it can be executed on the device or emulator. It also allows users to customize their test scripts for a particular UI. The user can generate UI-guided test inputs based on a state transition model generated on-the-fly.

Smart-Monkey [33] is an upgraded version of the Android Monkey tool, which is used to test Android apps and generate new test cases by combining both event-based testing elements and automatic random tests. It employs an extended FSCS-ART technique proposed by [27]. Test cases consist of a sequence of user events and system events based on the distance from the event sequence and ART (Adaptive Random Testing) used in other event-driven software. The strategy can reduce the

number of test cases and the time required to identify the first fault. Smart-Monkey creates a transition model of the app by using the random exploration approach before generating the test cases through the random walk.

Dynodroid [10] is a tool that generates input dynamically in an Android app. Dynodroid selects one relevant event to the app's current state before repeating the process. This tool generates "n" number of events one after another in an observe-select-execute cycle, where the first event is installed before the app is started in the emulator. Dynodroid supports three different heuristic exploration strategies, which include two different random techniques and an active learning technique at the execution of all the different events. The Dynodroid tool allows both automated and manual input generations.

ExtendedRipper [35] is an exploration-based technique that uses a dynamic analysis in the Android apps. In this technique, the event pattern includes numerous context events such as location change, GPS enable or disable, screen orientation, acceleration changes, and incoming calls or SMS. These event patterns are manually defined to generate test cases. In the literature, it was reported that ExtendedRipper achieved better code coverage than GUI Ripper [34] and AndroidRipper [11].

Android Monkey [20] also known as UI/App Exerciser Monkey is a black-box GUI testing tool in the Android SDK. Among the existing test generation tools, this random testing tool gained considerable popularity from society. Other than its simplicity, it has demonstrated good compatibility with a myriad of Android platforms which made it the most commonly used tool for numerous industrial applications. It is a command-line tool used directly in the device/emulator. It can generate pseudo-random events with unexpected scenarios to an AUT. It produces randomly generated events that serve as the test input in the absence of any guidance. Thus, the test exploration can be uniformly traversed throughout the GUIs (i.e., low activity coverage) and it cannot incorporate user-defined rules such as inserting a password or preventing logging out. Additionally, the generated events are low level with hard-coded coordinates, which complicates the reproduction and debugging processes [8]. Moreover, Android Monkey is unable to turn the sequence of events into test cases.

3.3. Grey Box Android Test Input Generation Tools

Stoat [12] performs a stochastic model testing in the following steps: (1) it creates a probabilistic model by exploring and analyzing the apps GUI interactions dynamically; (2) it optimizes the state-model by performing Gibbs sampling and directs test generation from the optimized model to achieve a higher code and activity coverage performance.

Sapienz [13] uses a multi-objective search-based testing approach to explore and optimize the test sequences automatically, minimize the test sequence length, and maximize the code coverage and fault detection. Sapienz combines search-based, random fuzzing, systematic exploration, and multi-level instrumentation. To explore the app components, it utilizes the specific GUIs and complex sequences of input events with a predefined pattern. This predefined pattern is termed as motif genes that capture the experience of the testers. Thus, it produces a higher code coverage by concatenating the atomic events.

CrashScope [40] can be used to discover, report, and reproduce crashes in Android apps. It uses a combination of static and dynamic analysis. Crashscope examines the Android app using a systematic input generation to detect a crash. As a result, it produces an HTML crash report which consists of screenshots, detailed crash reproduction steps, and a replay-able script.

A3E-Targeted [38] prioritizes the exploration of activities that can be reached from the initial activity of a static activity transition graph. The strategy is based on high-level control flow graphs that captured the activity transitions. It is constructed from the static dataflow analysis on the app's bytecode. It lists all of the activities before calling them in the absence of user intervention. However, it represents each activity as an individual state without considering its different states. This misleads the apps since not all states of the activities are explored. Moreover, it does not revisit old activities explicitly and may affect the exploration of new code which should be reached by different sequences.

4. Case Study Design

This analysis adopts the empirical case study method used in software engineering, as reported in [41,42]. The method includes three steps: (1) specify case study objectives, (2) selecting a case study with data collection, and (3) case study design for executing and evaluating.

4.1. Case Study Objectives

The main question to answer from this experiment is how effective Android test input generation tools in detecting crashes? To answer the main question, the research questions of this study are as follow:

RQ 1. What is the method and activity coverage achieved by the test input generation tools?

RQ 2. How is the performance of the test input generation tools in detecting unique crashes?

RQ 3. How does the event sequence length affect the coverage and crash detection of the test input generation tools?

4.2. Case Study Criteria

Coverage criterion is one of the critical testing requirements that some elements of the app should be covered [43]. A combination of different granularities from method and activity coverage is essential to achieve better testing results for Android apps. The activities and methods are the central building elements of the apps, thus the numeric values of the activity and method coverage are intuitive and informative [38]. Activity is the primary interface for user interaction. An activity consists of several methods and underlying code logic. Hence, improvement of method coverage ensures most of the app's functionalities associated with each activity are explored and tested [38,39]. Moreover, activity coverage is a prerequisite condition to reveal crashes that might happen during the interaction with the app's UI. The more coverage a tool explores, the higher the chances a potential crash can be found [44]. In this study, the number of inputs generated by a tool within a time limit was measured.

C1. Method Coverage (MC): MC is the ratio of the number of methods called during execution of the AUT to the total number of methods in the source code of the app. By improving the method coverage, it is envisaged that most of the app's functionalities are explored and tested [8,38,39,44].

C2. Activity Coverage (AC): AC is defined as the ratio of activities explored during the execution to the total number of activities present in the app. A high activity coverage value indicates that a greater number of screens have been explored, and thus it will be more exhaustive for the app exploration.

C3. Crash detection: Crashes lead to termination of the app's processes and dialogue is displayed to notify the user about the app crash. The more code the tool explores, the higher the chances it discovers a potential crash.

4.3. Subjects Selection

For the experimental analysis, 50 Android apps were chosen from F-Droid [45] and AppBrain [46] repositories. Table 2 lists the type of apps according to the app category, the number of activities, methods, and line of code in the app (which offers a rough estimate of the app size). These apps were earmarked from the repositories based on three features:

(1) the app's number of activities: the apps were categorized by a small (number of activities less than five), medium (number of activities less than ten), and a large (number of activities more than ten). In total, 27 apps were selected for the small group, while 17 apps were screened for the medium group. Lastly, six apps were added to the large group. The app's activities were determined in the Android manifest file of the app.

(2) user permissions required: in this study, only apps that require at least two of the user permissions were selected to evaluate how the tools react to different system events. These permissions

include access to contacts, call logs, Bluetooth, Wi-Fi, location, and camera of the device. The app permissions were determined either by checking the manifest file of the app or by launching the app for the first time and viewing the permissions request(s) that popped up.

(3) version: only apps that are compatible with Android version 1.5 and higher were selected in this study.

Table 2. Overview of Android apps selected for testing.

No	APP Name	Package Name	Version	Category	Activity	Methods	LOC
1	Aard	aarddict.android	1.5	Books	6	438	65,511
2	Open Document	at.tomtasche.reader	1.6	Books	3	446	13,796
3	Bubble	com.nkanaev.comics	4.1	Books	2	463	58,397
4	Book Catalogue	com.eleybourn.bookcatalogue	2.1	Books	21	1548	11,264
5	Klaxon	org.nerdcircus.android.klaxon	1.6	Communication	6	162	5733
6	Sanity	cri.sanity	2	Communication	28	1398	44,215
7	WLAN Scanner	org.bitbatzen.wlanscanner	4	Communication	1	141	5441
8	Contact Owner	com.appengine.paranoid_android.lost	1.5	Communication	2	79	2502
9	Divide	com.khurana.apps.divideandconquer	2.1	Education	2	195	25,284
10	Raele.concurseiro	raele.concurseiro	3	Education	2	92	1309
11	LolcatBuilder	com.android.lolcat	2.3	Entertainment	1	79	578
12	MunchLife	info.bpace.munchlife	2.3	Entertainment	2	39	163
13	Currency	org.billthefarmer.currency	4	Finance	5	148	5202
14	Mileage	com.evancharlton.mileage	1.6	Finance	50	2486	92,548
15	TimeSheet	com.tastycactus.timesheet	2.1	Finance	6	198	7126
16	Boogdroid	me.johnmh.boogdroid	4	Game	3	398	3726
17	Hot Death	com.smorgasbork.hotdeath	2.1	Game	3	365	28,104
18	Resdicegame	com.ridgelineapps.resdicegame	1.5	Game	4	144	2506
19	DroidWeight	de.delusions.measure	2.1	Health and Fitness	8	411	13,215
20	OSM Tracker	me.guillaumin.android.osmtracker	1.6	Health and Fitness	8	346	49,335
21	Pedometer	name.bagi.levente.pedometer	1.6	Health and Fitness	2	244	6695
22	Pushup Buddy	org.example.pushupbuddy	1.6	Health and Fitness	7	165	4602
23	Mirrored	de.homac.Mirrored	2.3	Magazines	4	219	825
24	A2DP Volume	a2dp.Vol	2.3	Navigation	8	641	23,294
25	Car cast	com.jadn.cc	1.5	Music and Audio	12	459	18,127
26	Ethersynth	net.sf.ethersynth	2.1	Music and Audio	8	168	1208
27	Jamendo	com.teleca.jamendo	1.6	Music and Audio	13	1046	30,444
28	Adsdroid	hu.vsza.adsdroid	2.3	Productivity	2	1215	5080
29	Maniana	com.zapta.apps.maniana	2.2	Productivity	4	891	28,526
30	Tomdroid	org.tomdroid	1.6	Productivity	8	840	29,147
31	Talalarmo	trikita.talalarmo	4	Productivity	3	387	1350
32	Unit	info.staticfree.android.units	1.6	Productivity	3	547	22,993
33	Alarm Clock	com.angrydoughnuts.android.alarmclock	2.7	Productivity	5	676	2453
34	World Clock	ch.corten.aha.worldclock	2.3	Productivity	4	315	1156
35	Blockinger	org.blockinger.game	2.3	Puzzle	6	356	2000
36	OpenSudoku	cz.romario.opensudoku	1.5	Puzzle	10	444	24,601
37	Applications info	com.majeur.applicationsinfo	4.1	Tools	3	323	3614
38	Dew Point	de.hoffmannsgimmickstaupunkt	2.1	Tools	3	75	4791
39	drhoffmann	de.drhoffmannsoftware	1.6	Tools	9	164	896
40	FindMyphone	se.erikofsweden.findmyphone	1.6	Tools	1	2969	4056
41	List my Apps	de.onyxbits.listmyapps	2.3	Tools	4	96	4262
42	Sensors2Pd	org.sensors2.pd	2.3	Tools	4	621	16,625
43	Terminal Emulator	jackpal.androidterm	1.6	Tools	8	994	24,930
44	Timeriffic	com.alfray.timeriffic	1.5	Tools	7	709	28,956
45	Addi	com.addi	1.1	Tools	4	2133	133,448
46	Alogcat	org.jtb.alogcat	2.3	Tools	2	199	846
47	Android Token	uk.co.bitethebullet.android.token	2.2	Tools	6	288	3674
48	Battery Circle	ch.blinkenlights.battery	1.5	Tools	1	81	251
49	Sensor readout	de.onyxbits.sensorreadout	2.3	Tools	3	683	6596
50	Weather	ru.gelin.android.weather.notification	2.3	Weather	7	695	19,837

5. Case Study Execution

In Table 1, only available tools were selected. The tools were Sapienz [13], Stoat [12], Droidbot [36], Humanoid [37], Dynodroid [10], and Android Monkey [20]. These tools were selected due to their excellent ability in generating user and system events, which aim to increase the possibility of finding faults in system events. Moreover, Sapienz, Stoat, Android Monkey, and Dynodroid possessed the best code coverage and fault detection in continuous mode as compared to AndroidRipper [11], A3E [38], PUMA [47], and ACTEve [26] in previous evaluation [8,19]. All the tools were installed on a dedicated machine before starting the experiments. Android emulator x86 ABI image (KVM powered) was used for the experiments. Due to the inherent poor performance of Android emulators on ARM, it was excluded since it could affect the test input generation tools negatively.

Android emulator was used due to its compatibility with Sapienz and Dynodroid. In contrast, Stoat, DroidBot, Humanoid, and Android Monkey support both emulators and real devices. Moreover, Android SDK version 4.4.2 (Android KitKat, API level 19) was configured in Sapienz, Stoat, Droidbot, Humanoid, and Android Monkey because Sapienz supports Android KitKat only. For Dynodroid, SDK version 2.43 (Android Gingerbread, API level 10) was used. The Android emulators were configured with 2 Gigabytes of RAM and 1 Gigabyte of SD card.

To achieve a fair comparison, a new Android emulator was configured for each run to avoid any potential side-effects that may occur between the tools and apps. As Dynodroid [10] was reported in the study, Android Monkey was set up to produce 20,000 inputs/hour. To avoid biased findings, other tools were run with their respective default configurations without any fine-tuning of the parameters. Each test input generation tool was allowed to run and execute tests for 60 min on each specified app. To compensate for the possible influence brought by randomness during testing, the test was run for triplicates (with each test consisting of one test generation tool and one applicable app that is being tested). Lastly, the final coverage and progressive coverage were recorded separately. An average value was calculated from the three tests and presented as the final results.

6. Case Study Results

Table 3 shows the results obtained from the six testing tools. Cells with a grey background indicate the maximum value achieved during the test. The percentage value is an average rounded-up value from the three tests iterations on each AUT.

Table 3. Results on method coverage, activity coverage, and crash detection by test input generation tools.

App Name	Method Coverage (%)						Activity Coverage (%)						# of Unique Crashes					
	Sa	St	Dr	Hu	M	Dy	Sa	St	Dr	Hu	M	Dy	Sa	St	Dr	Hu	M	Dy
A2DP Volume	53.6	55.8	49.7	60.3	39.3	0.0	100	100	100	100	71	0	1	0	2	2	0	0
Aard	17.4	16.4	16.4	16.2	17.4	17.4	33	33	33	33	33	33	1	0	1	1	0	0
Addi	10.1	9.5	9.6	9.6	4.6	4.7	50	50	50	50	50	25	1	1	1	1	1	1
Adsdroid	56.0	56.0	56.0	56.3	56.1	56.0	100	100	100	100	100	100	1	1	0	0	1	0
Alarm Clock	24.6	64.5	43.9	62.7	24.9	17.9	60	60	60	60	20	40	1	0	2	2	0	0
Alogcat	72.9	75.9	52.8	72.9	67.3	46.2	100	100	100	100	100	100	0	0	0	0	0	0
Android Token	54.5	57.6	54.5	49.5	51.4	50.0	67	67	50	67	50	50	0	0	0	0	0	0
applicationsinfo	64.5	64.3	64.4	64.7	44.3	44.3	100	100	100	100	67	67	0	0	0	0	0	0
Battery Circle	81.5	81.5	81.5	84.0	79.0	79.0	100	100	100	100	100	100	0	0	0	0	0	0
Blockinger game	77.5	81.7	81.5	80.1	16.6	11.2	100	100	100	100	67	50	0	0	0	0	0	0
Boogdroid	13.0	10.1	13.0	16.7	15.6	15.3	100	33	67	67	56	33	1	0	1	1	1	1
Book Catalogue	31.7	4.0	33.0	32.9	43.4	32.7	41	5	48	38	43	29	1	1	1	1	1	1
Bubble	54.9	55.9	36.9	30.5	67.6	0.0	100	100	100	50	100	0	2	2	0	0	1	0
Car cast	44.9	46.2	41.6	43.2	34.0	29.8	75	67	67	67	47	72	2	3	1	1	2	1
Contact Owner	54.4	54.4	57.0	57.0	51.1	51.9	50	50	50	50	50	50	1	1	1	1	1	1
Currency	58.8	64.0	40.3	56.8	42.6	40.5	100	100	80	100	100	60	1	1	0	0	0	0
Dew Point	75.6	73.3	76.4	77.3	58.7	58.7	100	100	100	100	67	67	2	1	1	1	0	1
Divide	52.8	47.2	52.8	52.8	74.4	46.2	100	100	100	100	100	100	0	0	0	0	0	0
DroidWeight	74.1	62.9	69.8	70.6	72.7	72.7	50	38	67	75	58	38	1	0	0	0	0	0
drhoffmann	55.9	59.8	51.2	59.1	43.3	36.6	85	100	93	93	93	78	2	2	1	2	2	1
Ethersynth	66.7	65.1	50.0	57.7	66.7	64.3	100	100	100	100	88	63	1	1	0	0	0	0
FindMyphone	0.1	0.1	0.1	0.1	0.1	0.1	100	100	100	100	100	100	0	0	0	0	0	0
Hot Death	74.6	59.4	65.8	70.4	76.8	55.6	100	100	100	100	100	100	0	1	0	0	0	0
Jamendo	53.1	34.7	43.9	43.9	24.4	0.0	62	31	38	38	46	0	2	1	1	1	0	0
Klaxon	44.4	38.3	36.6	35.8	39.9	41.4	83	83	83	83	83	78	0	0	0	0	0	0
List my Apps	76.4	75.3	46.9	72.9	72.9	72.9	50	100	25	100	100	25	0	0	0	0	0	0
LolcatBuilder	32.9	27.8	32.9	32.9	25.3	25.3	100	100	100	100	100	100	0	0	1	1	1	0
Maniana	75.4	58.4	54.1	54.0	72.8	66.9	75	75	75	75	75	50	0	0	1	1	0	0
Mileage	30.7	16.0	25.6	25.5	25.0	27.6	44	25	38	40	22	27	1	1	1	1	1	1
Mirrored	33.3	39.7	47.0	47.0	32.4	30.1	75	75	75	75	50	50	1	0	0	0	0	0
MunchLife	66.7	66.7	66.7	66.7	59.0	59.0	100	100	100	100	100	100	0	0	0	0	0	0
Open Document	54.4	42.2	45.3	45.3	70.6	36.8	33	33	33	33	33	33	0	0	0	0	0	0
OpenSudoku	62.5	54.5	40.3	40.3	42.2	38.1	50	30	30	30	50	50	1	1	1	1	1	1
OSM Tracker	44.2	62.9	48.1	48.3	50.6	43.1	75	71	83	88	92	50	1	2	1	1	1	1
Pedometer	70.1	69.7	61.5	61.5	83.2	70.4	100	100	50	100	100	50	1	2	0	0	1	0
Pushup Buddy	48.5	48.9	48.9	56.4	44.2	43.0	57	57	43	71	57	43	0	0	0	0	0	0
Raele.concurseiro	41.7	42.1	41.7	41.7	41.7	41.7	100	100	100	100	100	100	0	0	0	0	0	0
Resdicegame	62.5	48.6	62.5	53.5	47.9	47.9	100	100	100	100	100	100	0	0	0	0	0	0
Sanity	28.4	18.6	24.9	24.3	30.4	14.2	48	29	48	46	61	21	1	1	0	0	0	0
Sensor read out	30.2	30.0	30.0	30.0	27.8	30.0	100	100	67	67	67	67	1	0	0	0	0	1
Sensors2Pd	17.1	20.0	20.5	21.7	20.5	20.5	100	100	100	100	100	100	2	1	1	1	1	0

Table 3. *Cont.*

App Name	Method Coverage (%)						Activity Coverage (%)						# of Unique Crashes					
	Sa	**St**	**Dr**	**Hu**	**M**	**Dy**	**Sa**	**St**	**Dr**	**Hu**	**M**	**Dy**	**Sa**	**St**	**Dr**	**Hu**	**M**	**Dy**
Talalarmo	88.1	88.1	82.2	88.9	90.4	71.9	100	100	100	100	100	100	0	0	0	0	0	0
Terminal Emulator	55.2	44.2	51.3	51.7	52.3	49.5	38	38	38	38	38	25	1	1	0	0	0	0
Timeriffic	63.8	50.5	57.1	57.1	59.8	54.9	86	52	57	57	71	29	0	0	0	0	0	0
TimeSheet	59.4	33.5	27.8	27.8	20.7	20.2	100	67	50	50	50	56	0	0	0	0	0	0
Tomdroid	36.1	51.9	37.5	37.0	40.8	0.0	63	63	75	75	63	0	0	0	0	0	0	0
Unit	69.9	55.6	52.5	52.5	69.1	54.8	33	33	33	33	67	33	0	0	0	0	0	0
Weather notifications	59.4	50.0	41.7	37.4	74.2	67.6	71	52	57	57	71	43	1	0	0	0	1	0
WLAN Scanner	66.0	63.1	65.2	65.2	61.7	61.7	100	100	100	100	100	100	0	0	0	0	0	0
World Clock	43.8	56.8	29.2	29.2	22.9	22.9	100	100	75	75	50	50	0	0	0	0	0	0
Overall	39.8	35.1	36.1	36.8	36.9	28.8	66.3	55.3	60.7	62.8	58.1	42.0	32	25	19	20	17	11

Keywords: Sa: Sapienz, St: Stoat, Dr: Droidbot, Hu: Humanoid, M: Android Monkey, Dy: Dynodroid.

RQ1. What is the method and activity coverage achieved by the test input generation tools?

(1) Method coverage: the method coverage was collected from Ella [48]. Ella is a binary instrumentation tool for Android apps. From Table 3, one can be seen that Sapienz achieved the best method coverage on 14 out of the 50 apps. It is also important to mention that our result matches that reported by Mao et al., [13] in 2016. It outperformed the other tools due to its multi-level instrumentation approach that provided the traditional white-box coverage and Android user interface coverage. The instrumentation refers to the technique that modifies the source code or the byte code at the compile time to track the execution of the code at runtime. Sapienz used EMMA [49] white-box instrumentation tool to achieve full statement coverage, while Ella [48] exploited a black-box instrumentation tool for method coverage. Next, Android Monkey had the second-best performance with the highest method coverage in nine out of the 50 apps. Android Monkey adopted a random exploratory strategy that allowed more inputs to be generated. On the contrary, Humanoid achieved a lower coverage value of 36.8% as compared to Android Monkey with a coverage value of 36.9%. This can be ascribed to the ability of Humanoid in prioritizing critical UI elements. On average, other tools such as DroidBot, Stoat, and Dynodroid achieved a method coverage of 36.1%, 35.1%, and 28.8%, respectively. DroidBot can quantify the efficacy of the test without the source code or instrumentation. An outlier was observed in an AUT (Book catalog app had a total method number of 1548, in which Stoat only recorded an average of 4%) during testing, which was believed to have gradually affected its overall average method coverage.

Figure 2 presents the boxplots where the subscript x indicates the mean value of the final method coverage across the target apps. The boxes offer the minimum, mean, and maximum coverage achieved by the tools. This analysis revealed that all the tools were unable to cover more than 51% of the mean method coverage values.

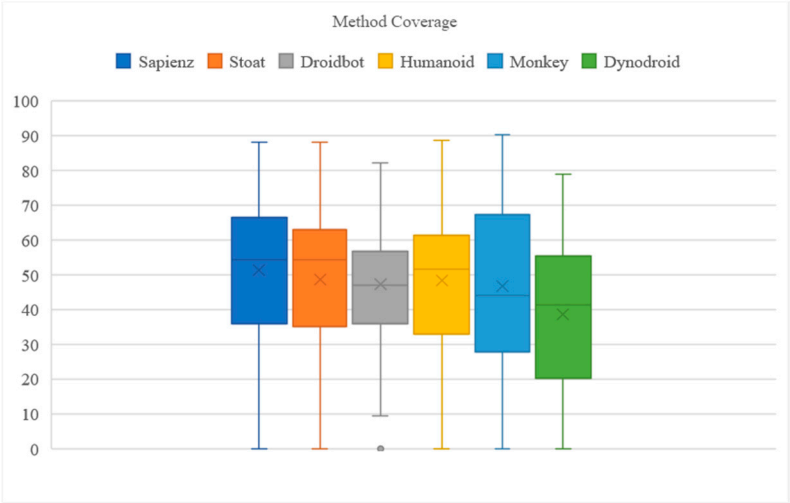


Figure 2. Variance of method coverage achieved across apps and three runs.

On average, both Sapienz and Android Monkey were observed to perform better than other tools. The other tools achieved a reasonably low level of method coverage. There are apps for which all the tools, including the best-performed tool, achieved shallow coverage, i.e., lower than 5%. An example is the FindMyPhone app. It was highly dependent on several external factors, such as the availability of a valid account. Furthermore, these inputs were almost impossible to generate automatically, and every tool stalled at the beginning of the exploration. Moreover, Dynodroid tools provide an initial option to manually interact with an app and allow the tool to perform the successive test input generation. Nonetheless, the features were excluded for two reasons: (1) poor scalability, and (2) an unfair advantage.

Figure 3 reports the progressive coverage of each tool over the time threshold of 60 min. The progressive average coverage of each of the test input generation tool was calculated across all 50 apps for every 20 min. The final coverage achieved was compared and reported. In the first 20 min, the coverage for all testing tools was observed to be increased rapidly as the apps were just started. At 40 min, the method coverage of many testing tools had been increased except for Android Monkey. The random approach of Android Monkey generated many redundant events, and these redundant events produced insignificant coverage when the time budget increased. In the end, Sapienz attained the highest method coverage after approximately 60 min of execution.

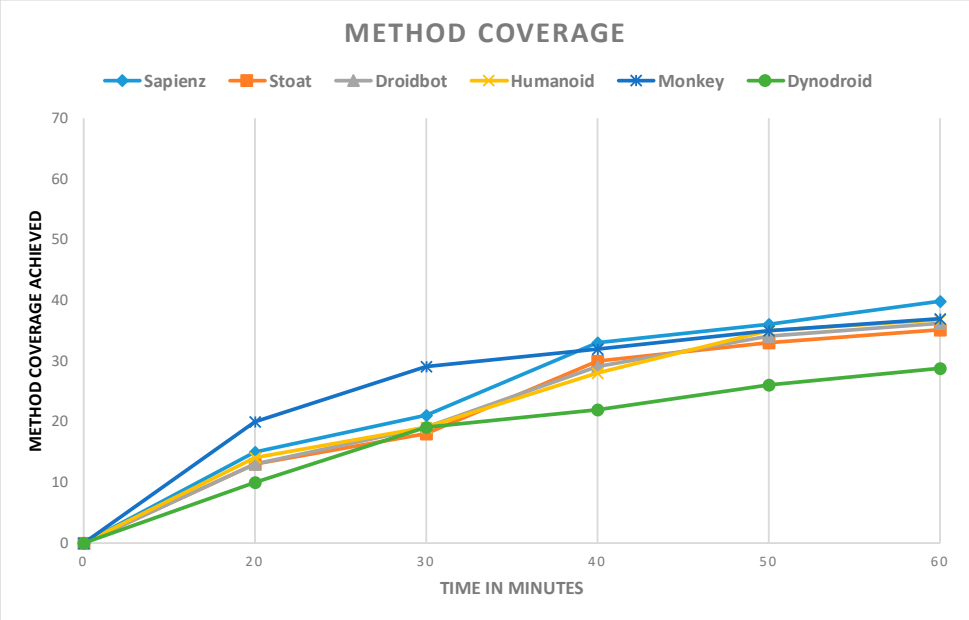


Figure 3. Progressive method coverage achieved across apps and three runs.

(2) Activity coverage: the activity coverage was measured intermittently by observing the activity stack of the AUT and recording all of the activities that have been listed down in the Android manifest file. The test input generation tools demonstrated much better activity coverage than the method coverage. From the results, Sapienz outperformed the other tools, similar to the previous experiment on method coverage. Due to its ability to explore and optimize the test sequences as reported by [13], Sapienz achieved the best mean activity coverage in six out of the 50 apps with an overall average activity coverage value of 66.3%. Following that, Humanoid was the second-best test input generation tool in the context of activity coverage. Humanoid performed a sequence of meaningful actions, which was opposite to Android Monkey's inability to test new core functionality. Therefore, activity coverage was prioritized in Android Monkey. Despite this, Android Monkey produced more inputs than other approaches, and it was highly limited in its random approach. Sapienz, Stoat, and Humanoid were able to achieve 100% activity coverage in 20 apps. DroidBot demonstrated the best coverage in the Book catalog app as compared to other tools in the present study. It integrated a simple and yet effective depth-first exploration algorithm, which pruned the UI components to have an event. In contrast, Stoat and Dynodroid achieved much lower coverages than the other tools, with an overall average of 55.3% and 42.0%, respectively. This is because Stoat had an internal null intent fuzzing, which directly started the activities with empty intents. There was an outlier in one of the AUTs (Mileage) among the F-Droid apps, whose total activity was 50 activities. Therefore, the causes of such uncovered app's activities were manually investigated from the test input generation tools. The Mileage app contains activities that required text inputs to fill up the text fields before allowing access to the next activity. During execution, Sapienz, Stoat, and Android Monkey produced random text inputs. While DroidBot and Humanoid created text input fields by searching for a sequence of predefined inputs. Dynodroid

paused the test for manual inputs after a text input field like logging in password is required. However, none of the test tools were able to explore more than 44% of activity coverage on the Mileage app.

Figure 4 reports the variance of the mean coverage of three runs across all 50 apps. The horizontal axis shows the tools used and the vertical axis indicates the percentage of coverage. The boxes show the minimum, mean, and maximum coverage achieved by the tools.

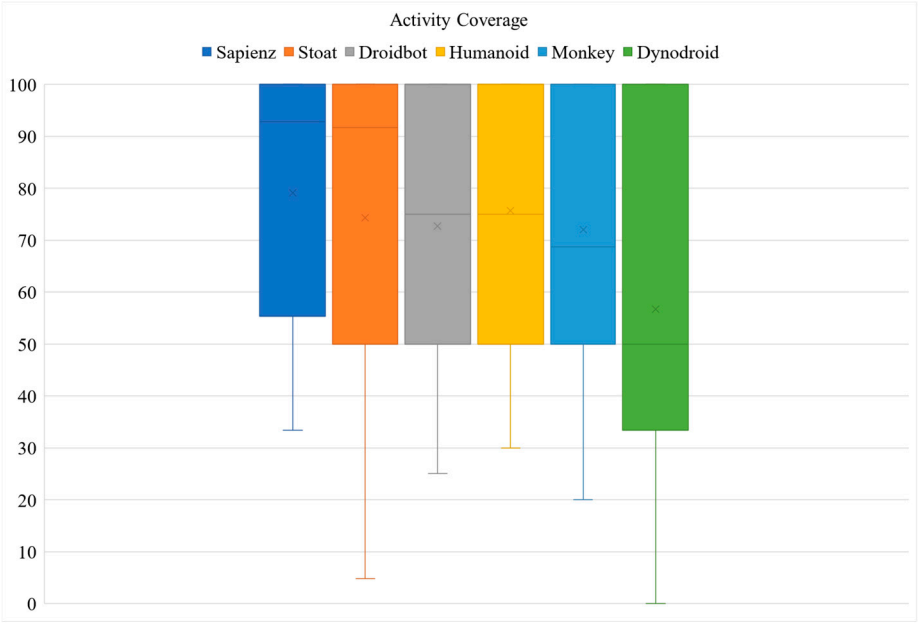


Figure 4. Variance of Activity coverage achieved across apps and three runs.

From Figure 4, one can observe that the activity coverage was higher than the method coverage. Sapienz, Stoat, DroidBot, Humanoid, Android Monkey, and Dynodroid obtained a coverage percentage increase of 100% with a mean coverage of 79%, 74%, 73%, 76%, 72%, and 57%, respectively. All tools were able to cover more than 50% of the activity coverage. From the results, it was found that 25 out of the 50 apps were not fully covered. In some apps, reaching activity requires a unique path of activity transitions from the root to the target activity or the activity that requires the filling of correct text inputs. Thus, it is recommended to support predefined test inputs as implemented in DroidBot and Humanoid. Moreover, some activities require a particular system event, such as connecting to Bluetooth. Hence, it is essential to generate guided system events instead of random generation of system events. To overcome such a problem, one possible solution is to instrument an Android system event related to the AUT. One can conclude that the guided test input generation approaches implemented in Sapienz, Stoat, DroidBot, and Humanoid were more effective than the random approaches as the latter requires a longer time to cover all activities which could be impractical in large apps with complex GUIs. Furthermore, more sophisticated test generation approaches are more effective due to the built model heuristics that generate high coverage tests.

As shown in Figure 5, the activity coverage for all testing tools increased with time until a point of convergence. The average convergence time of the tools was about 50 min, but the fastest convergence of each tool was different. From the results, Android Monkey, Humanoid, and Sapienz had the highest coverage at 20, 40, and 50 min, respectively.

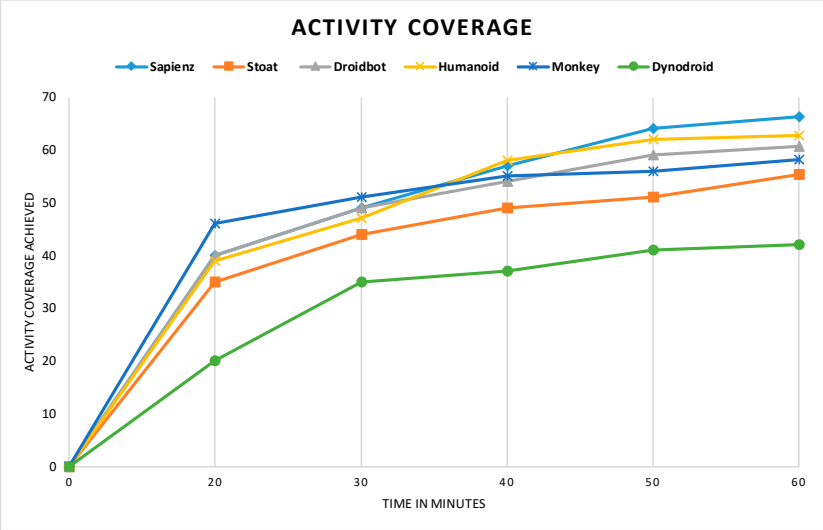


Figure 5. Progressive activity coverage achieved across apps and three runs.

Other tools such as Stoat, DroidBot, and Dynodroid did not achieve the highest coverage before the final convergence time of 60 min. Stoat required more execution time because it has an initial phase to construct an app state-model for the test case generation. This indicated the significance of each tool in measuring the activity coverage of an AUT and synonymously checking the capacity to detect a crash. Our test evaluation also revealed that there were no significant variations between the Android Monkey random approach and other tools. Thus, the tools required a longer execution time to improve their coverage. Android Monkey explored the same activities repeatedly for a long time since it triggered events on random coordinates of the screen, and it has no knowledge of the location of widgets on a screen. Compared to Humanoid and DroidBot, both tools explored all of the components available in the activity. Therefore, both did not reach the deep activities in one of the AUTs (Jamendo) within the time budget.

RQ2. How is the performance of the test input generation tools in detecting unique crashes?

During testing, AUT entered a new state, i.e., the app encountered a fatal exception or became non-responsive. App crashes are usually interpreted as the end state/last state because the app fails to proceed with the execution. This section aims to detect and record all of the unique app crashes encountered by each test tool during the testing process. Each unique app crash has a different error stack that defined the error location. The data logs of the six tools were evaluated, collected, and compared to evaluate the effectiveness of each test tool.

For the testing process, LogCat [50] was used to check the crashes encountered repeatedly during the execution of the AUT. LogCat is a tool that uses the command-line interface to dump a log of all system-level messages. The system-level messages include error messages, warnings, system information, and debugging information. Each unique crash exception of the tool was recorded, and the execution process was repeated three times to prevent randomness in the results. The number of unique app crashes was used as a measure of the tool performance in detecting the crashes. To identify the unique crashes from the error stack, the logs were analyzed manually by following the Su et al. [12] protocol. To exclude the crashes that were unrelated to the app's execution, only the app's package name, filter crashes of the tool themselves, and the initialization errors of the apps in the Android emulator were retained. Next, a hash was computed over the sanitized stack trace of the crash to identify the unique crashes. Different crashes have different stack traces and thus a different hash. A recent study [37] has highlighted that crashes caused by the Android system or the test harness itself should not be counted because most of them were false positives. Thus, such crashes can be identified by checking the corresponding stack traces. In the literature, different studies have used the number of

unique crashes detected as the primary evaluation criteria. The higher the crash number detected (in comparison to other testing tools), the better the tool performance in detecting app crashes [44].

Figure 6 shows the distribution of crashes in each testing tool. Among all the six testing tools, Sapienz detected the highest number of unique app crashes. Sapienz outperformed the other tools because it used a Pareto-optimal Search-Based Software Engineering (SBSE) approach [51]. However, Sapienz used the Android Monkey input generation, which continuously generated events without waiting for the effect of the previous event. Sapienz triggered many ClassCastException and ConcurrentModificationException, where all of them were found via a trackball and directional pad events. However, these crashes were insignificant because trackballs and directional pads were generally unavailable on modern Android phones. Besides, Sapienz triggered numerous SQLiteExceptions on the Jamendo app for all three runs. The exceptions were mostly about querying on multiple non-existent tables in the app's SQLite database. As the apps rely heavily on the SQLite database and do not handle related exceptions adequately, these fatal SQL queries are frequently triggered by multiple locations of the app and caused different stack traces. None of the other tools tested in this experiment detected SQLiteExceptions. The reason is that triggering SQLiteExceptions require specific preconditions (e.g., forcibly terminating the app during initialization which involves SQL operations for creating these tables) that other tools might not create. Moreover, Android Monkey detected some stress testing bugs such as IllegalStateExceptions from the synchronizations between ListViews and their data adapters, IllegalArgumentExceptions from the mismatches of service binding or nonbinding due to the rapid switches of activity lifecycle callbacks, and OutOfMemoryErrors [12]. OutOfMemoryErrors may occur when the app attempts to load a large-size file from the SD card without user permission. Some exceptions can be detected under special configurations depending on the granted permissions, such as granting permission to access SD card [52].

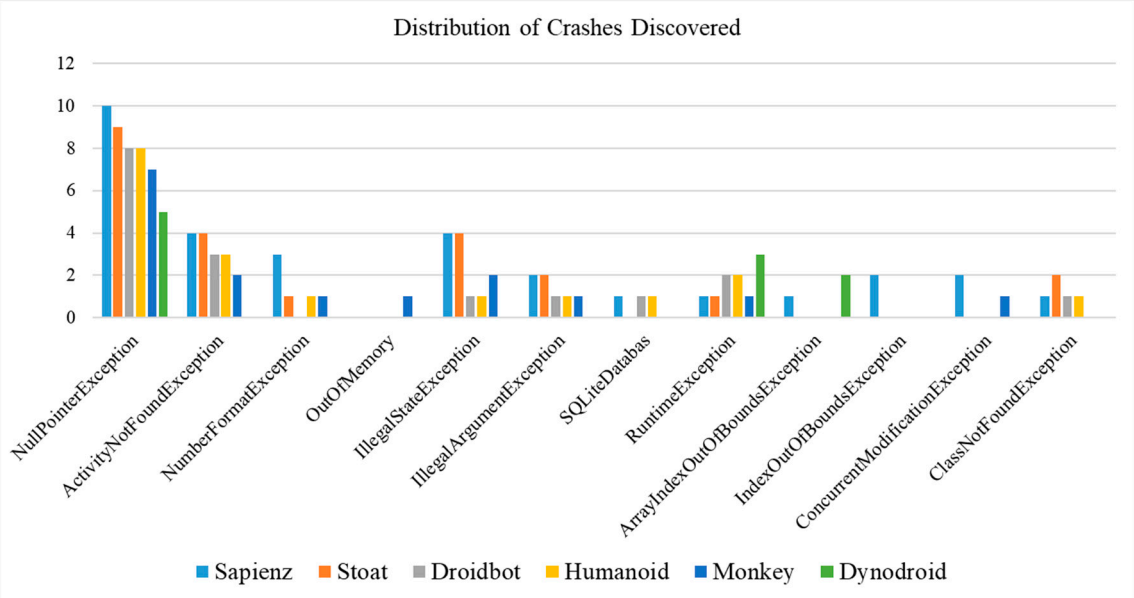


Figure 6. Distribution of Crashes Discovered.

Stoat was the second-best test input generation tool as it detected 25 unique app crashes. It used a Gibbs sampling method as a guide for model-based testing. Compared to Android Monkey, Stoat demonstrated better crash detection performance by injecting system events during testing. Stoat also used optimization techniques to guide the test generation by capturing all possible events arrangement, which allowed it to reveal faults. Stoat triggered many NullPointerExceptions on the apps such as "Car cast" during the starting of activities that took an Intent as input. Moreover, Stoat detected many exceptions that did not terminate the app processes, e.g., window leaked exceptions. Meanwhile,

Humanoid, Droidbot, Sapienz, and Android Monkey triggered NumberFormatException in the "Droid weight" app, by inputting invalid text value. Dynodroid triggered other types of exceptions such as ArrayIndexOutOfBoundsException and NullPointerException.

RQ3. How does the event sequence length affect the coverage and crash detection of the test input generation tools?

Minimizing the total number of events in a test suite will reduce the testing time, effort, and number of steps required to replicate a crash significantly. However, test input generation tools tend to produce large test suites with thousands of test cases. Each test case usually contains tens to thousands of events (e1, e2,.., en). The length of test case is generally defined as the number of events in it. Such test suites are challenging to be incorporated into regression testing due to the long run time required. Regression testing should be fast so that allows the same test suite to be used repeatedly during the development.

In this work, Android Monkey generated 20,000 input data points in an hour and explored the same activities repeatedly with no new coverage. An example of AUTs (A2DP Volume) is presented in Table 3. Android Monkey clicked the back button to return to the main activity and the cycle repeats. Such repeated actions caused redundant explorations and occupy much of the exploration time and number of events.

On the other hand, Humanoid and DroidBot explored all activities in the A2DP Volume app within a time limit and produced a smaller number of events (1000 inputs). The approach from these tools guided the input and thus meaningful input events were generated. Besides, Sapienz coverage increased with the number of events during the initiation of the apps. While all UI states were new, they could not exceed the peak point at 40 min as seen in Figures 3 and 5. Hence, Sapienz explored visited states and generated more event sequences. Table 4 shows the maximum number of events sequences required by each tool to achieve the results.

Table 4. Experimental results to answer research questions.

Tools	Activity Coverage (%)	Method Coverage (%)	Number of Crashes	Max Events Number
Sapienz	66.3	39.8	32	30,000
Stoat	55.3	35.1	25	3000
Droidbot	60.7	36.1	19	1000
Humanoid	62.8	36.8	20	1000
Monkey	58.1	36.9	17	20,000
Dynodroid	42.0	28.8	11	2000

On average, Stoat, DroidBot, Humanoid, and Dynodroid generated a total of 3000, 1000, 1000, and 2000 events in an hour, respectively. Sapienz produced 30,000 events in an hour and optimized the events sequence length through the generation of 500 inputs per AUT state. Nevertheless, it created the largest number of inputs. Thus, one can conclude that a longer event sequence length did not improve the coverage. Moreover, Sapienz, Stoat, and Android Monkey attained the highest number of events. However, the coverage improvement was similar to Humanoid and DroidBot, which generated a smaller number of events. Both Humanoid and Droidbot generated 1000 events in an hour but achieved better activity coverage of Stoat, Android Monkey, and Dynodroid.

Our results showed that the sequence of long events performed better than the shorter events sequence. However, long events sequence offered a small positive effect on the coverage and crash detection. That was confirmed in the previous study by Xie and Memon [53]. Xie and Memon concluded that there was no significant difference between the long and short tests, but more extended tests can find additional faults that shorter tests cannot. Likewise, Bae et al. [54] showed that more extended tests performed better than shorter tests. However, longer event length only had a small positive effect on code coverage. As a whole, longer events sequences increased the coverage and

crash detection; however, more extended events sequences have many disadvantages such as high redundancy and high computational costs, and they can be difficult to interpret manually.

7. Discussion and Future Research Directions

From the result, one can deduce that the relationship between three primary parameters tested (method coverage, activity coverage, and crash detection) was not linear, i.e., more activities and methods explored did not reflect more app crashes will be detected. Moreover, the experiment results revealed that a combination of a search-based approach and a random approach is promising to achieve thorough app exploration. Besides, the adoption of reinforcement learning algorithms may generate effective events, which may reduce the redundant execution of events sequence. Although the test input generation tools have been shown to be significantly effective in crash detection, there are several avenues of future research, and functions that should be considered by other tools were highlighted.

7.1. Events Sequence Redundancy

Event sequence redundancy refers to test cases with similar steps. In many cases, it may have tests contained in other tests or tests with loops. A high redundancy affects the method coverage and activity coverage efficiency negatively, as the testing tool will take a longer time to obtain the same coverage than that with low redundancy. Furthermore, the capability to find faults will be reduced since the test suite tends to re-execute the same steps. It is essential to highlight that experiment specifically to verify redundancy in tests generated by these tools were exclude in this work. This limitation should be addressed in future work.

To avoid the execution of the same steps, Sapienz runs an optimization process with the highest number of crashes. Humanoid prioritizes critical UI elements to determine the inputs to execute and construct a state transition model to avoid re-entry of visited UI states. Stoat generates relevant inputs from a static and dynamic analysis by inferring events from the UI hierarchy and events listeners in the app code. DroidBot generates UI-guided test inputs based on the position and type of the UI elements that defined the static information which is extracted from APK (e.g., list of system events) and dynamic analysis to avoid re-entry of explored UI states. From the results, Android Monkey presented excellent results in the activity and method coverage. However, it presented a low number of crashes. This tool uses a random exploration strategy and is more prone to redundancy. On the contrary, Dynodroid uses a guided and random exploratory approach, in which most of the unacceptable events are discarded based on the GUI structure and registered event listeners in an app.

7.2. Events Sequence Length

The desired goal of software testing is to detect fault using the shortest possible event sequences within the shortest time and using the minimum efforts [13]. Developers may reject longer sequences because it is impractical to debug and also unlikely to occur in practice. The longer the event sequence, the less likely it will occur in practice. Generation of long event sequences in GUI testing usually leads to an increase in the testing space. Sapienz optimizes event sequence length at the testing time by detecting the highest number of crashes. However, it could not detect deep crashes because it needs to return the app to a new clean state before starting a new testing script.

7.3. Crashes Diagnose

In this work, none of the tools tested in this experiment can generate comprehensible crash reports. However, both Crashscope [40] and DroidWalker [55] were the tools that can generate reproducible test scenarios and were not tested in this work. These tools can generate a detailed test report which informs the interacted elements. This feature allows the developers to fix the faults since the test case can be reproduced manually and also allowing an easier debug. Crashscope records more contextual information about bug-triggering event sequences. However, it still cannot handle exception bugs caused by inter-app communications [52]. Crashscope is not an open-source tool and was excluded

from this study. Tools under this study did not provide a comprehensible report which made the fault hard to reproduce. Since most of the tools were based on non-deterministic algorithms, rerunning the tool may not catch the same bugs.

7.4. Reproducible Test Cases

The ability to create a reproducible test is essential for a test generation tool because a developer needs to reproduce the test cases. When a bug is detected, the developers usually need to reproduce the bug for at least twice before the error is fixed. Since several tools are based on stochastic test case generation approaches, one cannot assure that the tests can be re-generated by the tools. Some tools such as Android Monkey generate test cases that are infeasible to be reproduced by a human, which makes them the least favorable option. Thus, app developers are unable to reproduce the crash during the exploration, to conduct a regression test after fixing the bug, or to execute the same test under different environments. More research efforts are still required to reproduce the intended bug described in a crash report effectively and faithfully.

7.5. System Events

Android apps are context-aware because they can integrate contextual data from a variety of system events. Context-aware testing is an important issue, mobile devices usually enable rich user interaction inputs. These inputs are either UI user events or system events. This brings many difficulties in generating test inputs that can expose the app's faults from the user and system events effectively. It is important to discover the faults that are often reported in the bug reports of Android apps and appear when the app is impulsively solicited by system events. Android Monkey and Stoat generate random system events, while Humanoid and DroidBot send guided events. Even though the testing tools in this experiment generated system events such as click on home or back buttons by sending intent messages, one should include all systems events (e.g., Wi-Fi, GPS, Sensors). For future works, experiment tools with other apps will be attempted by checking the ability of these tools in detecting the crashes caused by various conditions of system events.

7.6. Ease of Use

Based on the author's experience in setting up each of the tools, the tools that required extra effort in terms of configuration were described. Android Monkey required the least effort during the configuration. It is the most widely used tool due to its high compatibility with different Android platforms. Followed by Dynodroid, whose running version was obtained from a virtual machine found on the tool's page. Dynodroid was designed to operate with a standard version of an Android emulator. It can perform an extensive setup before the exploration. Similar to Android Monkey, both DroidBot and Humanoid were easy to use and provided much-advanced features. On the other hand, Stoat and Sapienz required considerable effort to operate because both tools demanded hours for a configuration with an Android emulator. Moreover, existing test input generation tools for Android apps are typically impractical for developers to use due to the instrumentation and the platform required.

7.7. Access Control

Access control is one of the key aspects of software security [56]. Many access control mechanisms exist for selectively restricting access to a software system's security-sensitive resources and functionalities [57–59]. Mobile device resources can collect sensitive data and may expose the user to security and privacy risks if apps misuse them and without the user's knowledge. For instance, Android apps may access resources that are not needed for their primary function, for example using the Internet, GPS, camera, or access sensitive data such as location, photos, notes, contacts, or emails. Android uses a permission-based security mechanism to mediate access to sensitive data and potentially dangerous device functionalities. However, it is not always a straightforward task to properly use this mechanism. Android app's behavior may change depending on the granted

permissions; it needs to be tested under a wide range of permission combinations [58]. Test input generation tools are used to generate inputs that trigger actual app behavior (e.g., crash). Though, these tools could be improved to consider the behavior of user interface elements that access sensitive user data and device resources. More context-based access control mechanisms are required to restrict apps from accessing specific data or resources based on the user context.

7.8. Fragmentation

Fragmentation is one of the significant problems that Android developers continuously have to deal with it. The term fragmentation has been used to describe variability due to the diversity of mobile devices vendors. Test input generation tools for Android should support a variety of devices that have different hardware characteristics and use various releases of Android framework (API versions) so that developers could assure the proper functioning of their apps on nontrivial sets of configurations. Thus, Droidbot, Humanoid, and Android Monkey can run on multiple versions of Android framework. Configuration sets can be represented as a testing matrix combining several variations of devices and APIs. Other aspects have been shown to impact testing beyond those associated with fragmentation. These include orientation of the device (e.g., landscape or portrait), localization (which may load different resources), and permissions [60]. None of the tools tested in this experiment have considered these aspects explicitly. More studies are required to study the non-deterministic app behaviors.

8. Threats to Validity

In this study, there are internal and external threats to the validity associated with the results of our empirical evaluation. In terms of internal validity, the default emulator used was proposed by Sapienz and Dynodroid. The publicly available versions of Sapienz and Dynodroid were designed to operate with a standard version of the Android emulator. Another threat to the internal validity of our study was Ella's instrumentation effect, which may affect the integrity of the results. These could be due to the errors triggered by the incorrect handling of the binary code or by errors in our experimental scripts. To mitigate such risk, the traces of the sample apps were inspected manually.

External validity was threatened by the representativeness of the study to the real world. In other words, representativeness means how closely the apps and tools used in this study reflect the real world. Moreover, the generalizability of the results used a limited number of subject apps. To mitigate these, a standard set of subject apps was used in the experiment with different domains, including fitness, entertainment, and tool apps. The subject apps were selected carefully from F-Droid, which is commonly used in Android GUI testing studies. Section 4.3 explains the details of the selection process. Therefore, the test was not prone to selection bias. To reduce the aforementioned threats, experimental works with broad types of subjects should be performed on a larger scale in the future.

9. Conclusions

This paper provides an empirical evaluation of the effectiveness of test input generation tools for Android testing that supported system events generation on 50 Android apps. An experimental analysis was performed to investigate the effect of events sequence length on the method coverage, activity coverage, and crashes detection. The testing tools were evaluated and compared based on three criteria: method coverage, activity coverage, and their ability to detect crashes. From this study, it was concluded that a long events sequence led to a small positive effect on coverage and crash detection. Both Stoat and Android Monkey attained the highest number of events. However, coverage performance was similar to Humanoid and DroidBot which generated a smaller number of events. Moreover, this study showed that Sapienz was the best-performing tool that satisfies all three criteria. Despite Sapienz optimized events sequence length, it generated the highest number of events and it is unable to detect crashes that can only be reached from a long events sequence. Besides, Android Monkey was able to reveal stress testing crashes. However, it was limited to generate inputs relevant to the app, mainly due to its randomness in generating unreproducible events with long sequences.

Moreover, most of the tools were able to find a fault in the user events, and none of them was able to find a fault in a system event. Besides, test input generation tools generate random text inputs that impact their coverage performance, which could be fixed in the future by supporting text prediction or integrating other text input generation techniques. For future works, experiment tools with other apps will be attempted by checking the ability of these tools in detecting the crashes caused by the system events. Moreover, how the Android app testing tools obtain redundancy and how the redundancy affects the detection of defects in Android apps will be investigated as well.

Author Contributions: Conceptualization, Data curation, Formal analysis, Methodology, Resources, Software, Visualization, and Writing—original draft preparation: H.N.Y.; Supervision: S.H.A.H., R.J.R.Y., and M.H.; Writing—review and editing: H.N.Y., S.H.A.H., R.J.R.Y. and M.H.; Funding acquisition, M.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Malaysia Ministry of Education, grant number FRGS/1/2018/ICT04/UMS/02/2.

Acknowledgments: We gratefully thank Malaysia Ministry of Education for the Fundamental Research Grant Scheme provided to us numbered FRGS/1/2018/ICT04/UMS/02/2.

Conflicts of Interest: The authors declare that they have no conflict of interest.

References

1. Chaffey, D. *Mobile Marketing Statistics Compilation|Smart Insights*. 2018. Available online: https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ (accessed on 16 December 2019).
2. IDC. IDC—Smartphone Market Share—OS. Available online: https://www.idc.com/promo/smartphone-market-share (accessed on 16 December 2019).
3. TheAppBrain. Number of Android Applications on the Google Play Store|AppBrain. Available online: https://www.appbrain.com/stats/number-of-android-apps (accessed on 16 December 2019).
4. Packard, H. *Failing to Meet Mobile App User Expectations: A Mobile User Survey*. Technical Report. 2015. Available online: https://techbeacon.com/sites/default/files/gated_asset/mobile-app-user-survey-failing-meet-user-expectations.pdf (accessed on 16 December 2019).
5. Ammann, P.; Offutt, J. *Introduction to Software Testing*; Cambridge University Press: Cambridge, UK, 2016.
6. Memon, A. *Comprehensive Framework for Testing Graphical User Interfaces*; University of Pittsburgh: Pittsburgh, PA, USA, 2001.
7. Joorabchi, M.E.; Mesbah, A.; Kruchten, P. Real Challenges in Mobile App Development. In Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 10–11 October 2013; pp. 15–24.
8. Choudhary, S.R.; Gorla, A.; Orso, A. Automated Test Input Generation for Android: Are We There yet?(e). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 429–440.
9. Arnatovich, Y.L.; Wang, L.; Ngo, N.M.; Soh, C. Mobolic: An automated Approach to Exercising Mobile Application GUIs Using Symbiosis of Online Testing Technique and Customated Input Generation. *Softw. Pr. Exp.* **2018**, *48*, 1107–1142. [CrossRef]
10. Machiry, A.; Tahiliani, R.; Naik, M. Dynodroid: An Input Generation System for Android Apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 224–234.
11. Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; De Carmine, S.; Memon, A.M. Using GUI Ripping for Automated Testing of Android Applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 258–261.
12. Su, T.; Meng, G.; Chen, Y.; Wu, K.; Yang, W.; Yao, Y.; Pu, G.; Liu, Y.; Su, Z. Guided, Stochastic Model-Based GUI Testing of Android Apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 245–256.

13. Mao, K.; Harman, M.; Jia, Y. Sapienz: Multi-Objective Automated Testing for Android Applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 94–105.
14. Zhu, H.; Ye, X.; Zhang, X.; Shen, K. A Context-aware Approach for Dynamic GUI Testing of Android Applications. In Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference, Taichung, Taiwan, 1–5 July 2015; pp. 248–253.
15. Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; Ta, B.D.; Memon, A.M. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Softw.* **2014**, *32*, 53–59. [CrossRef]
16. Usman, A.; Ibrahim, N.; Salihu, I.A. Test Case Generation from Android Mobile Applications Focusing on Context Events. In Proceedings of the 2018 7th International Conference on Software and Computer Applications, Kuantan, Malaysia, 8–10 February 2018; pp. 25–30.
17. Arnatovich, Y.L.; Ngo, M.N.; Kuan, T.H.B.; Soh, C. Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising. In Proceedings of the 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), Hamilton, New Zealand, 6–9 December 2016; pp. 193–200.
18. Kong, P.; Li, L.; Gao, J.; Liu, K.; Bissyandé, T.F.; Klein, J. Automated testing of android apps: A systematic literature review. *IEEE Trans. Reliab.* **2018**, *68*, 45–66. [CrossRef]
19. Wang, W.; Li, D.; Yang, W.; Cao, Y.; Zhang, Z.; Deng, Y.; Xie, T. An Empirical Study of Android Test Generation Tools in Industrial Cases. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 738–748.
20. Google. UI/Application Exerciser Monkey|Android Developers. Available online: https://developer.android.com/studio/test/monkey (accessed on 10 December 2019).
21. Linares-Vásquez, M.; White, M.; Bernal-Cárdenas, C.; Moran, K.; Poshyvanyk, D. Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 111–122.
22. Google. Understand the Activity Lifecycle|Android Developers. Available online: https://developer.android.com/guide/components/activities/activity-lifecycle.html (accessed on 25 December 2019).
23. Memon, A. GUI testing: Pitfalls and process. *Computer* **2002**, *35*, 87–88. [CrossRef]
24. Yu, S.; Takada, S. Mobile application test case generation focusing on external events. In Proceedings of the 1st International Workshop on Mobile Development, Amsterdam, The Netherlands, 30 October–31 December 2016; pp. 41–42.
25. Rubinov, K.; Baresi, L. What Are We Missing When Testing Our Android Apps? *Computer* **2018**, *51*, 60–68. [CrossRef]
26. Anand, S.; Naik, M.; Harrold, M.J.; Yang, H. Automated Concolic Testing of Smartphone Apps. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Cary, NC, USA, 11–16 November 2012; p. 59.
27. Chen, T.Y.; Kuo, F.-C.; Merkel, R.G.; Tse, T. Adaptive Random Testing: The Art of Test Case Diversity. *J. Syst. Softw.* **2010**, *83*, 60–66. [CrossRef]
28. Wang, P.; Liang, B.; You, W.; Li, J.; Shi, W. Automatic Android GUI Traversal with High Coverage. In Proceedings of the 2014 Fourth International Conference on Communication Systems and Network Technologies, Bhopal, India, 7–9 April 2014; pp. 1161–1166.
29. Saeed, A.; Ab Hamid, S.H.; Sani, A.A. Cost and effectiveness of search-based techniques for model-based testing: An empirical analysis. *Int. J. Softw. Eng. Knowl. Eng.* **2017**, *27*, 601–622. [CrossRef]
30. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2011.
31. Méndez Porras, A.; Quesada López, C.; Jenkins Coronas, M. Automated Testing of Mobile Applications: A Systematic Map and Review. In Proceedings of the 2015 18th Conferencia Iberoamericana en Software Engineering (CIbSE 2015), Lima, Peru, 22–24 April 2015; pp. 195–208.
32. Google. Crashes Android Developers. Available online: https://developer.android.com/topic/performance/vitals/crash (accessed on 25 December 2019).
33. Haoyin, L. Automatic Android Application GUI Testing—A Random Walk Approach. In Proceedings of the 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), Chennai, India, 22–24 March 2017; pp. 72–76.

34. Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; De Carmine, S.; Imparato, G. A Toolset for GUI Testing of Android Applications. In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 650–653.
35. Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; Amatucci, N. Considering Context Events in Event-Based Testing of Mobile Applications. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 18–22 March 2013; pp. 126–133.
36. Li, Y.; Yang, Z.; Guo, Y.; Chen, X. DroidBot: A Lightweight UI-Guided Test Input Generator for Android. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20–28 May 2017; pp. 23–26.
37. Li, Y.; Yang, Z.; Guo, Y.; Chen, X. A Deep Learning Based Approach to Automated Android App Testing. *arXiv* **2019**, arXiv:1901.02633.
38. Azim, T.; Neamtiu, I. 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, Indianapolis, IN, USA, 26–31 October 2013; pp. 641–660.
39. Koroglu, Y.; Sen, A.; Muslu, O.; Mete, Y.; Ulker, C.; Tanriverdi, T.; Donmez, Y. QBE: QLearning-Based Exploration of Android Applications. In Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Luxembourg, 18–22 March 2013; pp. 105–115.
40. Moran, K.; Linares-Vásquez, M.; Bernal-Cárdenas, C.; Vendome, C.; Poshyvanyk, D. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, USA, 11–15 April 2016; pp. 33–44.
41. Kitchenham, B.A.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; El Emam, K.; Rosenberg, J. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.* **2002**, *28*, 721–734. [CrossRef]
42. Perry, D.E.; Sim, S.E.; Easterbrook, S.M. Case Studies for Software Engineers. In Proceedings of the 26th International Conference on Software Engineering, Edinburgh, UK, 23–28 May 2004; pp. 736–738.
43. Morrison, G.C.; Inggs, C.P.; Visser, W. Automated Coverage Calculation and Test Case Generation. In Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, Cape Town, South African, 14–16 September 2020; pp. 84–93.
44. Dashevskyi, S.; Gadyatskaya, O.; Pilgun, A.; Zhauniarovich, Y. The Influence of Code Coverage Metrics on Automated Testing Efficiency in Android. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2216–2218.
45. F-Droid. F-Droid—Free and Open Source Android App Repository. Available online: https://f-droid.org/ (accessed on 10 December 2019).
46. AppBrain. Monetize, Advertise and Analyze Android Apps|AppBrain. Available online: https://www.appbrain.com/ (accessed on 20 December 2019).
47. Hao, S.; Liu, B.; Nath, S.; Halfond, W.G.; Govindan, R. 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, Bretton Woods, NH, USA, 16–19 June 2014; pp. 204–217.
48. Saswat, A. *ELLA: A Tool for Binary Instrumentation of Android Apps*. 2015. Available online: https://github.com/saswatanand/ella (accessed on 10 November 2019).
49. Roubtsov, V. *Emma: A Free Java Code Coverage Tool*. 2005. Available online: http://emma.sourceforge.net/ (accessed on 10 November 2020).
50. Google. Command Line Tools|Android Developers. Available online: https://developer.android.com/studio/command-line (accessed on 16 December 2019).
51. Harman, M.; Mansouri, S.A.; Zhang, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv. (CSUR)* **2012**, *45*, 11. [CrossRef]
52. Su, T.; Fan, L.; Chen, S.; Liu, Y.; Xu, L.; Pu, G.; Su, Z. Why my app crashes understanding and benchmarking framework-specific exceptions of android apps. *IEEE Trans. Softw. Eng.* **2020**, 1. [CrossRef]
53. Xie, Q.; Memon, A.M. Studying the Characteristics of a "Good" GUI Test Suite. In Proceedings of the 2006 17th International Symposium on Software Reliability Engineering, Raleigh, NC, USA, 7–10 November 2006; pp. 159–168.

54. Baek, Y.-M.; Bae, D.-H. 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, Singapore, 3–7 September 2016; pp. 238–249.
55. Hu, Z.; Ma, Y.; Huang, Y. DroidWalker: Generating Reproducible Test Cases via Automatic Exploration of Android Apps. *arXiv* **2017**, arXiv:1710.08562.
56. Kayes, A.; Kalaria, R.; Sarker, I.H.; Islam, M.; Watters, P.A.; Ng, A.; Hammoudeh, M.; Badsha, S.; Kumara, I. A Survey of Context-Aware Access Control Mechanisms for Cloud and Fog Networks: Taxonomy and Open Research Issues. *Sensors* **2020**, *20*, 2464. [CrossRef] [PubMed]
57. Shebaro, B.; Oluwatimi, O.; Bertino, E. Context-based access control systems for mobile devices. *IEEE Trans. Dependable Secur. Comput.* **2014**, *12*, 150–163. [CrossRef]
58. Sadeghi, A.; Jabbarvand, R.; Malek, S. Patdroid: Permission-Aware GUI Testing of Android. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 220–232.
59. Borges, N.P.; Zeller, A. Why Does this App Need this Data? Automatic Tightening of Resource Access. In Proceedings of the 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), Xi'an, China, 22–27 April 2019; pp. 449–456.
60. Kowalczyk, E.; Cohen, M.B.; Memon, A.M. Configurations in Android testing: They Matter. In Proceedings of the 1st International Workshop on Advances in Mobile App Analysis, Montpellier, France, 4 September 2018; pp. 1–6.

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).