GRAPHICAL USER INTERFACE TEST CASE GENERATION FOR ANDROID APPS USING Q-LEARNING

HUSAM N. S. YASIN

FACULTY OF COMPUTER SCIENCE AND
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2021

GRAPHICAL USER INTERFACE TEST CASE GENERATION FOR ANDROID APPS USING Q-LEARNING

HUSAM N. S. YASIN

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2021

UNIVERSITY OF MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: Husam N. S. Yasin

Matric No: WHA130069

Name of Degree: Doctor of Philosophy (PhD)

Title of Project Paper/Research Report/Dissertation/Thesis ("this Work"):

Graphical User Interface Test Case Generation for Android Apps using Q-Learning.

Field of Study: Software Quality

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date: 27 April 2021

Subscribed and solemnly declared before,

Witness's Signature Date: 27 April 2021

Name:

Designation:

GRAPHICAL USER INTERFACE TEST CASE GENERATION FOR ANDROID APPS USING Q-LEARNING

ABSTRACT

Software testing is an effective means for assuring the quality of applications. Android applications (or mobile apps) have become an essential part of our daily life. Statistics affirm that 85% of smartphones worldwide are Android-based. Unfortunately, 17% of Android apps are still considered low-quality apps due to app crashes that can be avoided through intensive and extensive testing. Recently, Graphical User Interface (GUI) testing of Android app has gained considerable interest from the industries and research community due to its excellent capability to verify the operational requirements of GUI components. In the related literature, GUI test generation tools focus on generating tests and exploring app functions using different approaches. These tools are used to make the testing process more effective by finding faults; more comprehensive by achieving better code coverage and faster by producing the smallest possible event sequences. However, a common limitation of these tools is the low code coverage because of the tools' inability to find the right combination of actions that can drive the application into new and important states. Exploring the app's state extensively requires long event sequences to find the correct combination of actions, leading to excessively long transitions and wasted time. Such tools must choose not only which user interface element to interact with, but also which type of action to be performed to increase the percentage of code coverage and to detect faults with a limited time budget. This research addresses the problem of generating an effective test for Android apps that maximizes the instruction, method, and activity coverage by minimizing redundant execution of events. Hence, it proposes a Q-Learning-based test coverage approach developed in DroidbotX to generate GUI test cases for Android apps. It is a fully automated black-box testing approach that uses Upper Confidence Bound (UCB) exploration-exploitation strategy to generate actions that visit unexplored states of the app and uses the execution of the app on the generated actions to construct a state-transition model. Instead of randomly selecting the inputs, the test generator learns how to act in an optimal way that explores new states by using new actions to gain more rewards. Thus, the never selected actions can present a higher reward when compared to already executed actions. This reduces the redundant execution of events and increases coverage. The overall performance of DroidbotX was compared to five state-of-the-art test generation tools on 30 Android apps. DroidbotX achieved 51.5% accuracy for instruction coverage, 57% for method coverage, and 86.5% for activity coverage. It triggered 18 crashes within the time limit and shortest event sequence length compared to the other tools. The results demonstrate that the adaptation of Q-Learning with UCB exploration can significantly improve the effectiveness of the generated test cases. The computation time complexity of the Q-Learning-based test coverage algorithm was also analyzed. The results showed that time complexity was reduced significantly from $O(n^3)$ to $O(n^2)$. It is based on the average case scenario by randomly considering some of the events, states, and actions, and using the probabilistic distribution of states, actions, and events, and average them over time.

Keywords: Android, GUI Testing, Test case Generation, Reinforcement Learning, Q-Learning.

PENGHASILAN KES UJIAN GRAPHICAL USER INTERFACE UNTUK APLIKASI ANDROID MENGGUNAKAN Q-LEARNING

ABSTRAK

Pengujian perisian adalah kaedah yang berkesan untuk menjamin kualiti sistem-sistem aplikasi. Aplikasi Android (atau aplikasi mudah alih) telah menjadi sebahagian penting dalam kehidupan harian kita. Statistik mengesahkan 85% telefon pintar di seluruh dunia adalah berlandas-Android. Malangnya, 17% dari aplikasi Android boleh dianggap bekualiti rendah berpunca daripada keranapan aplikasi yang dapat dielakkan melalui pengujian intensif dan ektensif. Baru-baru ini, pengujian Antara Muka Grafik Pengguna (AGP) bagi aplikasi Anroid telah mendapat banyak perhatian dari komuniti industri dan penyelidikan berpunca dari keserlahan kemampuan untuk mengesahkan keperluan operasi untuk komponen AGP. Di dalam kajian kesusasteraan berkaitan, peralatan pengujian AGP tertumpu kepada penjanaan pengujian dan menerokai fungsian aplikasi melalui beberapa pendekatan. Peralatan ini digunakan untuk membina proses pengujian yang lebih efektif dengan mengenalpasti kesalahan; lebih menyeluruh untuk mencapai liputan kod yang lebih baik dan pantas dengan meminimumkan perlaksanaan peristiwa berurutan. Walaubagaimanapun, kelemahan am peralatan   ialah liputan kod yang rendah kerana ketidakbolehan peralatan mencari kombinasi yang tepat bagi aksi yang boleh memandu aplikasi kepada keberadaan baru dan penting. Penerokaan keberadaan aplikasi secara meluasnya memerlukan urutan peristiwa yang panjang untuk mencari kombinasi aksi-aksi yang tepat, mengheret secara meluasnya transisi panjang dan pembaziran masa. Peralatan ini perlu memilih bukan saja antara muka grafik pengguna tetapi juga aksi yang perlu dilaksana untuk menambah peratusan kod liputan dan untuk mengesan kesalahan dalam masa yang terhad. Penyelidikan ini bertujuan untuk menangani masalah dalam menghasilkan ujian yang berkesan untuk aplikasi Android yang memaksimumkan liputan arahan, aktiviti dan kaedah dengan meminimumkan pelaksanaan peristiwa yang berlebihan. Justeru itu, ia mencadangkan pendekatan liputan ujian berasaskan Pembelajaran-Q yang dibangunkan dalam DroidbotX untuk menghasilkan kes-kes ujian untuk aplikasi Android. Pendekatan ini adalah pedekatan pengujian kotak hitam automatik yang menggunakan strategi Batasan Keyakinan Atas *(BKA)* eksplorasi-eksplotasi untuk menghasilkan aksi yang dapat memeriksa keberadaan aplikasi yang belum diterokai dan menggunakan pelaksanaan aplikasi pada aksi yang dihasilkan untuk membina model peralihan-keberadaan. Sebalik memilih input secara rawak, penjana pengujian belajar untuk bertindak optimum untuk meneroka keberadaan baru menggunakan aksi baru untuk mendapat lebih ganjaran. Justeru, aksi yang tidak dipilih dipertengahkan dengan ganjaran yang lebih tinggi berbanding aksi yang telah dijana. Pendekatan ini dibina menjadi alat ujian yang dinamakan DroidbotX. Prestasi keseluruhan DroidbotX telah dibandingkan dengan lima peralatan pengujian yang canggih pada 30 aplikasi Android. DroidbotX telah mencapai ketepatan 51.5% untuk liputan arahan, 57% untuk liputan kaedah, dan 86.5% untuk liputan aktiviti. Ia telah mencetuskan 18 keranapan dalam had masa dan urutan peristiwa terpendek berbanding dengan peralatan lain. Keputusan ini menunjukkan bahawa penyesuaian *Q-Learning* dengan penerokaan UCB dapat meningkatkan keberkesanan kes ujian yang dijalankan secara ketara. Komputasi masa kerumitan berdasarkan pengujian liputan *Q-Learning* juga telah dianalisiss. Keputusan menunjukkan masa kerumitan dikurangkan secara signifikan dari $O(n^3)$ kepada $O(n^2)$. Ia berdasarkan scenario kes purata secara rawak mengambil kira beberapa peristiwa, keberadaan, dan aksi, dan menggunakan kebarangkalian taburan keberadaan, aksi, dan peristiwa dan purata dengan masa.

Kata kunci: *Android*, Pengujian AGP, Penjanaan kes ujian, Pembelajaran Pengukuhan, *Q-Learning*.

ACKNOWLEDGEMENTS

In the name of Allah, the Most Gracious and the Most Merciful

Alhamdulillah, all praises to Allah for the strength and His blessing in completing this thesis painstaking work.

My most sincere gratitude goes out to my supervisors, Dr. Siti Hafizah Ab Hamid, and Dr. Raja Jamilah Raja Yusof for their invaluable support, encouragement, supervision, and useful suggestions throughout this research.

I would also like to thank the Faculty of Computer Science and Information Technology, especially the department of software engineering for the many and various means of support offered, and the University of Malaya for providing resources and a stimulating intellectual atmosphere.

To the beacons of my life, my beloved family, thank you for always having faith in me. I dedicate this research to my Father, Prof. Naseem Yasin, and my mother. The values they have instilled in me and their love for me had made me what I am today. Special thanks to my wife for always being with me and encouraging me. It would be a miss on my part if I do not thank my friends who have always supported me throughout the years of my study.

TABLE OF CONTENTS

Abstract ii

Abstrak iv

Acknowledgements vi

Table of Contents vii

List of Figures xi

List of Tables xii

List of Abbreviations xiii

CHAPTER 1: INTRODUCTION 1

1.1 Background 1

1.2 Motivation 4

1.3 Problem Statement 5

1.4 Research Objectives 7

1.5 Research Questions 7

1.6 Research Scope 8

1.7 Research Methodology 8

1.8 Conclusion 12

CHAPTER 2: LITERATURE REVIEW 13

2.1 Android 13

2.1.1 Android Platform Architecture 14

2.1.2 Android Application Components 17

2.1.3 Android Activity Lifecycle 20

2.2 Graphical User Interface (GUI) Testing 22

2.2.1 GUI Testing on Android Application 24

2.2.2 Faults in Android GUI Testing 24
2.2.3 GUI Testing Frameworks for Android 25
2.3 State-of-the-Art GUI Testing Tools for Android 28
2.3.1 Taxonomy of GUI Testing Tools for Android 28
2.3.1.1 Automated Testing Activity 30
2.3.1.2 Approach 30
2.3.1.3 Evaluation Methods 34
2.3.1.4 Characteristics 35
2.3.2 Comparison of GUI Testing Tools for Android 36
2.3.2.1 Automated Testing Activities 36
2.3.2.2 Approach 38
2.3.2.3 Evaluation Methods of GUI Testing Tools for Android 44
2.3.2.4 Characteristics of GUI Testing Tools for Android 44
2.4 Comparison of Test Case Generation Approaches for Android 49
2.4.1 Record and Replay 49
2.4.2 Random based 50
2.4.3 Model-based 52
2.4.4 Active Learning 56
2.4.5 Systematic based 58
2.4.6 Search based 60
2.4.7 Reuse based 60
2.5 Research Gaps and Limitations 61
2.5.1 Reproducible Test Cases 61
2.5.2 Test Oracle 62
2.5.3 Test Input Generation 62
2.5.4 Test Coverage 63

2.5.5 Crashes Diagnose ... 63

2.5.6 Fragmentation ... 64

CHAPTER 3: EXPERIMENTAL ANALYSIS ON TEST CASE GENERATION TECHNOLOGIES ... 65

3.1 Case Study Design ... 65

3.1.1 Case Study Objectives ... 66

3.1.2 Case Study Criteria ... 66

3.1.3 Apps Selection ... 67

3.2 Case Study Execution ... 68

3.3 Results and Discussion ... 72

3.4 Research Problems Found ... 87

3.4.1 Events Sequence Redundancy ... 87

3.4.2 Events Sequence Length ... 88

3.4.3 System Events ... 88

3.4.4 Access Control ... 89

3.4.5 Ease of Use ... 90

3.5 Threats to Validity ... 90

3.6 Conclusion ... 91

CHAPTER 4: PROPOSED SOLUTION ... 92

4.1 Reinforcement Learning ... 92

4.1.1 Deep Q Network ... 96

4.1.2 Actor-Critic ... 96

4.1.3 State-Action-Reward-State-Action (SARSA) ... 97

4.1.4 Q-Learning ... 97

4.2 Automated GUI Testing with Reinforcement Learning ... 98

4.3 Justification of the Proposed Approach 102

4.4 Implementation of the Proposed Approach 103

4.4.1 States and Action Representation 106

4.4.2 Exploration Strategy 107

4.4.2.1 Observer and Rewarder 108

4.4.2.2 Action Selector 109

4.4.3 Test Case Generation 112

4.5 Significance of the Proposed Approach 114

4.6 Conclusion 115

CHAPTER 5: EVALUATION OF THE PROPOSED APPROACH 116

5.1 Case Study Design 116

5.1.1 Research Questions for the Proposed Approach Evaluation 116

5.1.2 Case Study Criteria 117

5.1.3 Apps Selection 118

5.2 Case Study Execution 120

5.3 Results and Discussion 121

5.4 Threats to Validity 143

5.5 Conclusion 144

CHAPTER 6: CONCLUSIONS AND FUTURE WORK 145

6.1 Restatement of Research Objectives 145

6.2 Research Contributions 148

6.3 Significance of the Work 151

6.4 Limitation and Future Work 152

6.5 List of Scholarly Publications 154

References 155

LIST OF FIGURES

Figure 1.1: Research Methodology 11

Figure 2.1: Android Architecture for System Application Software Stack 15

Figure 2.2: Android application package structure 18

Figure 2.3: Android Activity Lifecycle 21

Figure 2.4: Taxonomy of GUI testing tools for Android Applications 29

Figure 3.1: Variance of method coverage achieved across apps and three runs. 75

Figure 3.2: Progressive method coverage achieved across apps and three runs. 76

Figure 3.3: Variance of activity coverage achieved across apps and three runs. 78

Figure 3.4: Progressive activity coverage achieved across apps and three runs. 79

Figure 3.5: Distribution of Crashes Discovered. 81

Figure 4.1: Reinforcement learning mechanism 95

Figure 4.2: Overview of DroidbotX 105

Figure 4.3: An example of state and actions representation from Android app 107

Figure 4.4: A UI state transition graph from a real-world Android app. 113

Figure 5.1: Variance of instruction coverage achieved across apps and five runs. 123

Figure 5.2: Variance of method coverage achieved across apps and five runs 126

Figure 5.3: Variance of activity coverage achieved across apps and five runs 128

Figure 5.4: Distribution of crashes discovered. 130

Figure 5.5: Progressive coverage 134

Figure 5.6: Time Complexity Analysis 141

LIST OF TABLES

Table 2.1: Comparison of Automated testing activities...37

Table 2.2: Comparison of GUI test case generation tools 41

Table 2.3: Type of evaluation method of GUI testing tools 45

Table 2.4: Characteristics of GUI testing tools..48

Table 3.1: Overview of Android applications selected for testing.................................69

Table 3.2: Overview of Android test generation tools..72

Table 3.3: Statistics of results on apps by test generation tools understudy..................73

Table 3.4: Statistics of crash results on apps by test generation tools understudy 84

Table 3.5: Experimental results to answer research questions.......................................86

Table 4.1: Overview of Android test generation tools adopted RL 100

Table 5.1: Overview of Android apps selected for testing...119

Table 5.2: Results on instruction coverage by test generation tools............................122

Table 5.3: Results on method coverage by test generation tools.................................125

Table 5.4: Results on activity coverage by test generation tools.................................127

Table 5.5: Statistics of crash results on apps by test generation tools understudy 131

Table 5.6: Experimental results to answer case study questions. 132

Table 5.7: Statistics of models built by Droidbot, Humanoid, and DroidbotX 135

Table 5.8: Q-Learning-based test generation algorithm corresponds to time..............137

Table 5.9: Action selector function corresponds to time. 138

Table 5.10: Q-Learning-based test generation algorithm corresponds to time for average time analysis...140

LIST OF ABBREVIATIONS

A2C	:	Advantage Actor-Critic
A3C	:	Asynchronous Advantage Actor-Critic
AAPT	:	Android Asset Packaging Tool
ADB	:	Android Debug Bridge
API	:	Application Program Interface
APK	:	Android Application Package
ART	:	Adaptive random testing
AUT	:	Application Under Test
BFS	:	Breadth-first search
DFS	:	Depth-first search
DQN	:	Deep Q Network
DVM	:	Dalvik Virtual Machine
GUI	:	Graphical User Interface
JVM	:	Java Virtual Machine
MBT	:	Model-Based Testing
RL	:	Reinforcement Learning
SARSA	:	State-Action-Reward-State-Action
SDK	:	Android Software Development Kit
SBSE	:	Search-Based Software Engineering
UCB	:	Upper Confidence Bound
URI	:	Uniform Resource Identifier
UTG	:	User Interface Transition Graph
VM	:	Virtual Machine
XML	:	eXtensible Markup Language

CHAPTER 1: INTRODUCTION

This chapter is an overview of in-depth research undertaken in this thesis. It presents GUI testing for Android apps and the associated research problem. Herein, this study's aim and objectives are clarified and, as such, describes the methodology proposed to achieve the aim and objectives.

In this chapter, the outline is organized as follows: Section 1.1 introduces the study's background for undertaking this research. It also highlights the significance of the work. Section 1.2 is an introduction to the research motivation. The identified research problems to be addressed in this thesis are highlighted in Section 1.3. Section 1.4 presents the aim and objectives of this study. Section 1.5 presents research questions, followed by Section 1.6, which outlines the research scope. While Section 1.7 presents the proposed methodology. The chapter is concluded in Section 1.8.

1.1 Background

Software testing is an important and major area of software reliability. It deals with the probability that the software will not cause system failure for a specific time under the specified condition (Musa, 1987). Reliability is an important attribute of software quality in addition to other attributes such as usability, fault prediction, and performance (Taylor-Sakyi, 2016). Mobile app reliability is crucial in apps development. Mobile apps are everywhere and operate in complex environments. These mobile apps are developed under market pressure over time. Mobile apps work on a variety of platforms. Such platforms include Windows, iOS, and Android.

Over the years, Android, in particular, has gone through rapid growth and frequent updates. According to a previous report (Chaffey, 2018), the number of Android apps downloaded is increasing drastically over the years. Android has over 2 billion active devices monthly. It dominates over 85% of the global market share (IDC, 2019). Google

Play Store is an Android apps' official market with more than 3.3 million apps, covering more than 30 categories, such as entertainment and personalization apps to education and financial apps. Concurrently, Android apps are updated on average every 60 days. A previous study (Statista, 2019) indicated that a mobile device, on average, has between 60 and 90 apps installed. Besides, an Android user, on average, spends 2 hours and 15 minutes on apps every day. With these statistics, unfortunately, in December 2019, it was reported by AppBrain that about 17% of the Android apps were low-quality apps (TheAppBrain, 2019). Furthermore, it was also reported that 53% of users would avoid using an app if the app crashed (Packard, 2015). The inferior quality of Android apps can be attributed to insufficient testing due to its rapid development practice and the fragmentation of mobile devices that have different hardware characteristics and use various releases of the Android framework (Canfora et al., 2016). Android developers tend to disregard good testing practices as it is considered time-consuming, expensive, and with a lot of repetitive tasks. The quality of apps depends on both functional (e.g., app crashes and unresponsive apps) and non-functional requirements such as the absence of failures and performance. Some of the major issues that heavily disrupt users' experience include app crashes and unresponsive apps, as reported by users review (Khalid et al., 2014). Mobile app crashes are evitable and avoidable by intensive and extensive testing of mobile apps. Android apps can be tested with a graphical user interface (GUI) testing tools to verify the app's functionality, usability, and consistency before they are released to the market (Ammann & Offutt, 2016; Joorabchi et al., 2013). To start Android apps testing, test cases will be generated with a series of events sequence on the GUI components to reveal faults. By faults, it means the inability of the app to perform required functions, which may or may not lead to a crash. The sequences of events (or test input) can be either from user interaction or system interaction. For example, a user interaction (or actions) usually involves clicking, scrolling, or typing texts into a GUI element like a button, image, or text block. While interaction with system includes receiving SMS notifications, app notifications, or phone calls.

The development of GUI test cases usually takes a lot of time and effort because of their non-trivial structures and the highly interactive nature of GUIs (Banerjee et al., 2013). Android apps usually possess numerous states and transitions, which can lead to an arduous testing process and poor testing performance. For the past decade, Android test generation tools have been developed to automate user interaction and allow system interaction as inputs (Amalfitano, Fasolino, Tramontana, De Carmine, & Memon, 2012; Amalfitano et al., 2014; Machiry et al., 2013; K. Mao et al., 2016; Su et al., 2017; Zhu et al., 2015). The focus of these tools is to generate test cases and explore the app's functions by employing different techniques. These techniques can be distinguished according to the way they generate tests (Linares-Vásquez et al., 2017); Random testing generates a randomized series of events sequence to trigger crashes; Model-based testing technique uses a directed graph-based model to correlate the relationship of the user interaction and the GUI of the apps; Record and replay testing record user interactions to generate repeatable scripts; Systematic based generates guided tests based on program analysis, and Q-Learning based. These techniques are implemented in sophisticated approaches (Koroglu et al., 2018; Machiry et al., 2013; K. Mao et al., 2016; Su et al., 2017) which are well presented in chapter 2.

Q-Learning is a type of model-free technique of reinforcement learning (Kaelbling et al., 1996). It was used in software testing in the past and has shown its ability to improve random-based techniques (Adamo, Khan, et al., 2018; Koroglu et al., 2018; Mariani et al., 2011; Vuong & Takada, 2018). It uses trial-and-error interactions to experience the consequences of actions. Initially, the Q-Learning agent interacts with the App Under Test (AUT) to identify the best action (from a set of actions available in the current state) that are most likely to discover unexplored app's states (Watkins & Dayan, 1992). Actions never selected can present a higher reward than actions already executed, which reduces the redundant execution of events and increases coverage. Accordingly, the knowledge of AUT is updated to find a policy that facilitates systematic exploration to make efficient future action selection decisions. This exploration interacts with AUT to construct a state transition model and generates test cases. These test cases follow the sequences of events that are the most likely to explore the app's functionalities. To conclude, for an effective test case, it is vital to generate effective actions first that reach new and important app states, which in turn leads to an increase in the coverage and fault detection.

1.2 Motivation

According to a dimensional research survey (Packard, 2015), 61 percent of users expect mobile apps to start in four seconds, while 49 percent want input responses within two seconds. Besides, if an app freezes, crashes, or has errors, 53 percent of users will uninstall it. Hence, effective approaches for testing Android apps are needed. GUI test case generation can be demanding because of many competing properties that developers care about such as code coverage, test sequence length, and the ability to detect faults (K. Mao et al., 2016). An analysis reveals that out of 600 open-source Android app projects hosted on the F-Droid repository (F-Droid, 2010), only 14 percent contain test cases, and about 9 percent have executable test cases with code coverage of over 40 percent (Kochhar et al., 2015). Coverage is an important metric to measure the effectiveness of testing (Memon et al., 2001). High code coverage is necessary for automated testing for the sake of minimizing human efforts and maximizing effectiveness (Wang et al., 2014). Due to the low code coverage of current approaches, there is still a need for continued study (Amalfitano, Fasolino, Tramontana, De Carmine, & Imparato, 2012; Choi et al., 2013; Hu & Neamtiu, 2011; Machiry et al., 2013).

1.3 Problem Statement

Regardless of the widespread use of Android apps, software reliability problems remain prevalent. Mobile apps developers heavily rely on user reviews to get reports on software crashes (Khalid et al., 2014; Linares-Vasquez et al., 2015; Palomba et al., 2015). The most common problems reported by mobile app users are app crashes (Khalid et al., 2014). App crashes give users an unsatisfactory experience, and negatively impact the app's overall rating (Khalid et al., 2014; Martin et al., 2016). An app is tested with an automatically generated sequence of events that simulates user interaction with the GUI which serves as test cases for executing system tests. Given unlimited time, to achieve high code coverage, all possible event sequences interaction and combinations can be tested. In practical situations where testing time is often limited, and the AUT contains a large number of possible combinations of interactions in each state and transitions between them, testing all possible scenarios is time-consuming and ineffective for large systems. Automated testing tools often choose a small subset of interaction sequences to explore, leaving many app functions unexplored. Thus, such tools must choose not only which GUI component to interact with, but also which type of action to perform. Each type of action on each GUI component is likely to improve the percentage of code coverage and fault detection. Automated testing tools are used to make the testing process more effective by finding faults; more comprehensive by achieving better code coverage and faster by producing the smallest possible event sequences (Dashevskyi et al., 2018). Even though there are existing studies to achieve these goals (Choi et al., 2013; Machiry et al., 2013; K. Mao et al., 2016; Su et al., 2017; Wang et al., 2014), however, these research tools encounter two major problems.

The first problem is low code coverage, existing tools are still insufficient to exercise the app thoroughly and thus cannot achieve high code coverage in automated testing (Choudhary et al., 2015; Wang et al., 2018). In particular, existing tools cannot effectively manage the exploration of states and as well minimize unnecessary transitions between them. Besides, existing tools cannot effectively explore a wide range of app functionalities because some of the app's functions can only be explored through a specific sequence of events. For instance, Android Monkey is the most commonly used tool for numerous industrial apps. It is regarded as the current state of practice for automated Android testing (Mahmood et al., 2014; Wang et al., 2018). However, Android Monkey requires more time to generate a long sequence of events. These events include redundant events that are repeatedly jumping between app activities and non-relevant to the current state that clicks on a non-interactive area on the screen (Clapp et al., 2016). These redundant events have no consistent pattern and cannot keep track of executed events.

For the second problem, 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. Minimizing the total number of events in a test suite will reduce the testing time, effort, and the number of steps required to replicate a crash significantly. Developers may reject longer sequences because it is impractical to debug and less likely it will occur in practice (K. Mao et al., 2016). However, test generation tools tend to generate large test suites with thousands of test cases. Each test case usually holds tens to thousands of events. The length of the 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. The generation of long event sequences in GUI testing usually leads to an increase in the testing space. For example, for a 10-event GUI, the number of all possible length-10 test cases is 1010. Even when considering possible restrictions on the combinations, the number might still be large.

1.4 Research Objectives

This research aims to generate effective GUI test cases for Android apps. Hence, this research proposes an approach that dynamically produces GUI test cases based on a Q-Learning technique. This approach systematically selects events and guides the exploration to explore the functionalities of an AUT to maximize instruction, method, and activity coverage by minimizing redundant event execution. The aim is accomplished by achieving the following objectives:

- To review the current state-of-the-art GUI testing tools to generate test cases for Android apps.
- To examine the effectiveness of test generation tools for Android apps in terms of method coverage, activity coverage, and crash detection.
- To develop an approach using Q-Learning to optimize test case generation that maximizes instruction coverage, method coverage, and activity coverage.
- To evaluate the ability of the proposed approach to generate effective test cases that detect crashes and maximize instruction coverage, method coverage, and activity coverage on real-world Android apps.

1.5 Research Questions

For this research, the main research questions are:

RQ1: What are the trends and future direction in state-of-the-art GUI testing tools for Android apps?

RQ2: How is the performance of state-of-the-art GUI test generation tools for Android apps in terms of event sequence length on the overall activity coverage, method coverage, and crash detection?

RQ3: How can Q-Learning be used to improve event sequence selection for optimizing test case generation?

RQ4: What is the effectiveness of the proposed approach in terms of instruction coverage, method coverage, activity coverage, and crash detection?

1.6 Research Scope

The scope of this research is to maximize instruction coverage, method coverage, and activity coverage by minimizing redundant execution of events. Meanwhile, this research excludes text prediction when sending text input and fault revelation, which requires generating an automated oracle.

1.7 Research Methodology

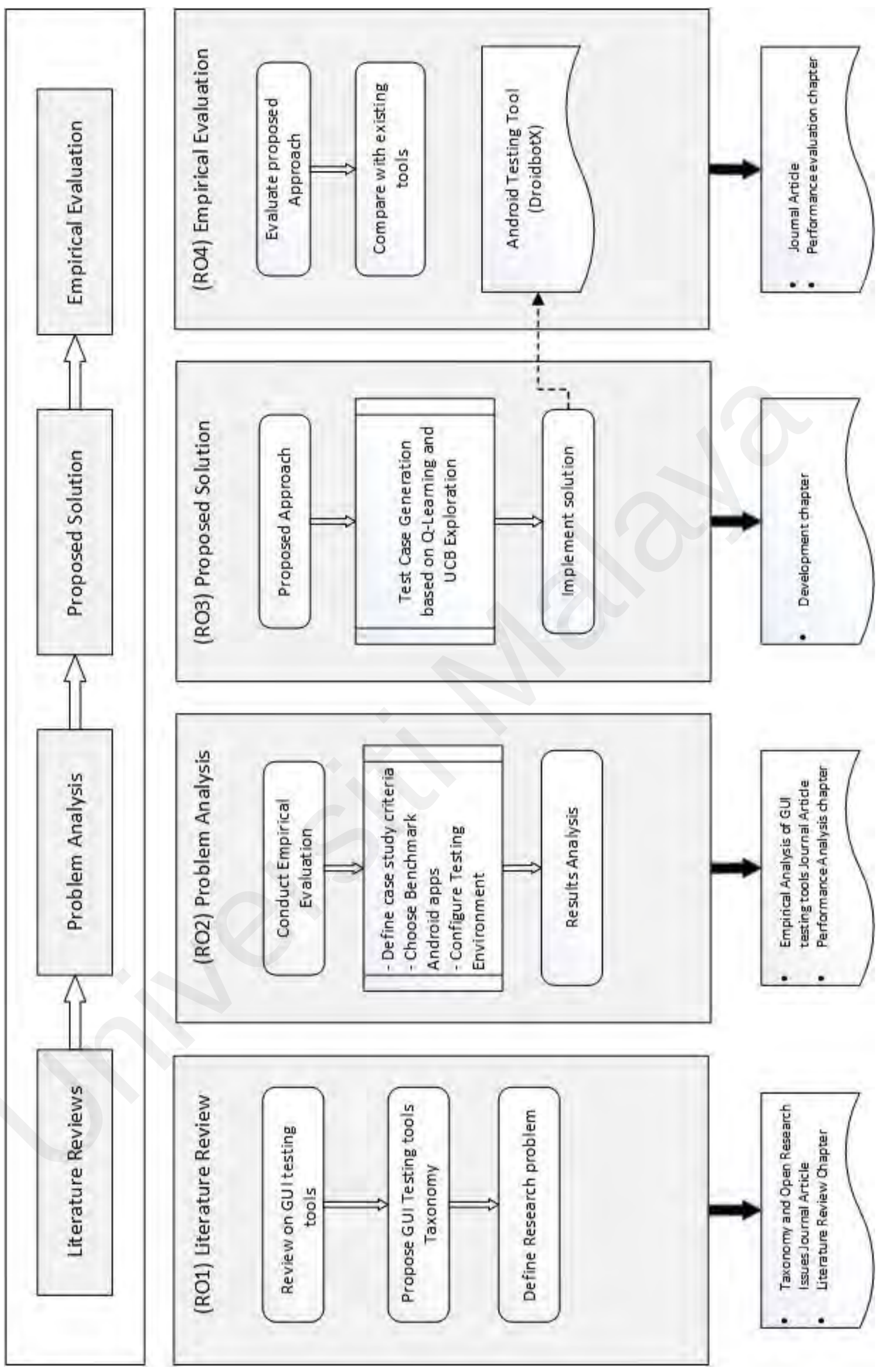
To achieve the aim and objective of this research, the steps shown in Figure 1.1 were followed.

A comprehensive review of the state-of-the-art GUI testing tools for Android apps was undertaken to analyze their strengths and weaknesses; a taxonomy was proposed to explore shared traits and contrasts among studied tools. The taxonomy was created by analyzing 45 different studies from 2011 to 2019. The studies were collected from five known data sources include ACM, IEEE Xplore, springer, science direct, and google scholar. The taxonomy contains four prominent parameters including (i) automated testing activities, (ii) GUI testing tools approach, (iii) evaluation methods (iv), and characteristics. Moreover, a comparison of GUI testing tools for Android apps was discussed and classified based on seven main approaches include (record and replay, random based, model-based, active learning, systematic based, search-based, and reuse based). Furthermore, several research issues in GUI testing for Android apps were identified through literature. The problems to be addressed in this thesis were also identified through literature. The research issues include reproducible test cases, test oracle, test input generation, test coverage, crash diagnosis, and fragmentation.

The problems identified were investigated, and their significance was thoroughly verified through empirical evaluation. This analysis employs the empirical case study method that is used in software engineering. The effectiveness of the test generation tools, especially in the event sequence length of the overall code coverage and crash detection, was evaluated on 50 Android apps. The event sequence length generally shows the number of steps required by the test 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. The test generation tools were evaluated and compared based on the activity coverage, method coverage, and capability in detecting crashes. Furthermore, several research problems in test generation tools for Android apps were identified. The issues to be addressed in this thesis were also investigated. The research issues include events sequence redundancy, event sequence length, system events, access control, and ease of use.

To alleviate the identified problems, a fully automated black-box testing approach based on the Q-Learning technique was proposed. A Q-Learning-based test coverage approach uses Upper Confidence Bound (UCB) exploration-exploitation as a learning policy, to create an efficient exploration strategy for GUI testing. The exploration strategy systematically selects events and guides the exploration toward revealing the functionalities of an AUT. It interacts and explores the app's functionalities following the strategy of observe-select-execute, where all the GUI actions of the current state of AUT are observed; one action is selected based on the selection strategy under consideration, and the selected action is executed on the AUT. Instead of randomly selecting the actions, the proposed approach learns how to act in an optimal way that explores new states by using new actions to gain more rewards. Thus, actions never selected can present a higher reward when compared to already executed actions, which reduce the redundant execution of events and increase instruction coverage, method coverage, and activity coverage. The proposed approach was implemented into a test tool named DroidbotX. DroidbotX constructs a state-transition model of the app and generates test cases. These test cases follow the sequences of events that are the most likely to explore the app's functionalities.

The performance of the proposed approach was evaluated via an empirical case study analysis. The overall performance of the proposed approach was compared to five state-of-the-art test generation tools. Five tools with different techniques have been chosen for the experiment as follows Sapienz (search-based), Stoat (model-based), Droidbot (model-based), Humanoid (deep Q network), and Android Monkey (random-based). These tools are the most recent techniques for Android testing. Thirty real-world Android apps were used in this evaluation chosen from the F-Droid repository. Instruction coverage, method coverage, activity coverage, crash detection, and time to run have been opted as performance metrics in this evaluation. The standard setup of the experiments was applied. The performance evaluation results were validated in comparison with the five state-of-the-art test generation tools results. Moreover, the computation time complexity of the Q-Learning test coverage algorithm was analyzed. The results show the significant performance of the proposed approach.



Literature Reviews
Problem Analysis
Proposed Solution
Empirical Evaluation
(RO1) Literature Review
Review on GUI testing tools
Propose GUI Testing tools Taxonomy
Define Research problem
Taxonomy and Open Research Issues Journal Article
Literature Review Chapter
(RO2) Problem Analysis
Conduct Empirical Evaluation
- Define case study criteria
- Choose Benchmark Android apps
- Configure Testing Environment
Results Analysis
Empirical Analysis of GUI testing tools Journal Article
Performance Analysis chapter
(RO3) Proposed Solution
Proposed Approach
Test Case Generation based on Q-Learning and UCB Exploration
Implement solution
Development chapter
(RO4) Empirical Evaluation
Evaluate proposed Approach
Compare with existing tools
Android Testing Tool (DroidbotX)
Journal Article
Performance evaluation chapter

Figure 1.1: Research Methodology

1.8 Conclusion

This chapter provides an overview of software testing and the difficulties that developers encounter to meet the users' demands. Thus, the app quality is often compromised due to poor app testing. This research proposed a Q-Learning-based test coverage approach developed as a tool called DroidbotX. The proposed approach aims to generate GUI test cases for Android apps to achieve maximum instruction coverage, method coverage, and activity coverage while minimizing redundant execution of event sequences. Chapter two provides comprehensive literature on Android apps, existing GUI testing tools, and its taxonomic parameters. In chapter three, the empirical evaluation of existing tools was conducted. The remaining chapter focuses on the in-depth explanation of the proposed approach for the effective generation of test cases.

CHAPTER 2: LITERATURE REVIEW

This chapter reviews the state-of-the-art GUI testing tools comprehensively for Android apps to analyze their strengths and weaknesses. A comprehensive thematic taxonomy is proposed based on an extensive review of the existing GUI testing tools for Android apps. The critical features and related aspects of these tools are thoroughly examined to evolve the proposed taxonomy. The tools are exhaustively analyzed according to the taxonomy parameters to explore shared traits and contrasts among existing tools. Finally, several research issues in Android app GUI testing are put forward that require further consideration to enhance the tools.

The chapter is organized: Section 2.1 presents an overview of the Android platform, its architecture, and its components. Section 2.2 discusses GUI testing, its faults, and frameworks. Section 2.3 proposes the thematic taxonomy of GUI testing tools for Android apps and compares the GUI testing tools for Android apps based on taxonomy parameters. Section 2.4 discusses the test case generation approaches for Android apps. Last, section 2.5 presents research gaps and limitations.

2.1 Android

Android is a mobile operating system that was released in 2008 to the market. It was developed by Google Inc. thenceforth, there had been a steady increase in the success rate of the Android platform. With its design, it has become the most popular mobile system in 2011, which has an open-source framework, a Linux-based layered software as compared to its competitors such as iOS (Apple) and Windows Phone (Microsoft).

According to the market research report from Statista (Statista, 2019), nearly 2.2 million apps are available in the Apple App Store, while Google Play remains the largest app store with accessible 3.3 million apps to its consumer. The popularity of Android among developer communities can be attributed to its open Java programming language-based development framework, as well as the availability of libraries with various functionalities (Li et al., 2016).

2.1.1 Android Platform Architecture

Android is an open-source OS built based on a consolidated Linux kernel created for an increasingly wide range of hardware and devices. Figure 2.1 illustrates the high-level architecture of the Android platform, the modified Linux kernel acts as a Hardware Abstraction Layer (HAL) and offers device driver, process and memory management, and networking capabilities, respectively. The library layer is configured with Java (which deviates from the conventional Linux design). It is in this layer that the special libc (bionic) for Android is located. The surface manager controls the user interface (UI) windows. Additionally, the Android runtime layer contains both the Dalvik Virtual Machine (DVM) and the core libraries (such as Java or IO). The core libraries provide the majority of the functionalities available in Android.

Android is focused on improving technology based on minimal resources available on mobile devices. Besides, the Android-specific application framework was developed and used to enhance the operating environment. The architectural framework behind Android is called the Android software stack, which consists of layers (Google, 2019h). They are as follows.

Linux Kernel. This layer is the core of the architecture of Android. It is founded on the Linux kernel with special additions for a model embedded platform. It provides the following functions: power management, memory management, process management, Hardware drivers, and security. It is used for better communication in software and hardware binding.

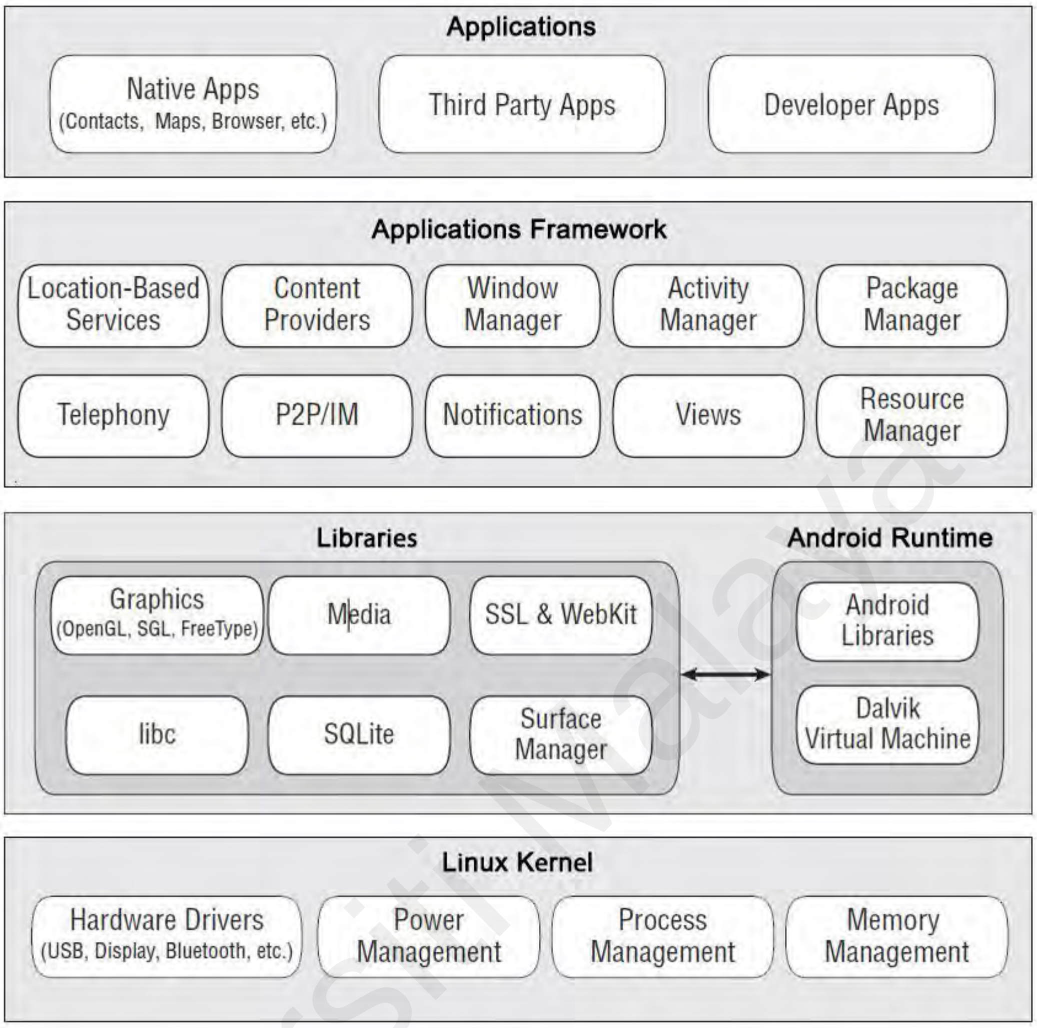


Figure 2.1: Android Architecture for System Application Software Stack

Hardware abstraction Layer. HAL consists of multiple library modules and is developed using native technology (C/C++ and shared libraries), each of which implements an interface for a specific type of hardware component. HAL lays out the standard interfaces which reveal the capabilities of the device's hardware to the higher-level Java API (Application Programming Interface) framework. The Android system downloads the library module for that hardware component when the API framework calls for hardware device access.

Dalvik Virtual Machine (DVM). Generally, Android-based systems employ their virtual machine (VM) known as a DVM that utilizes a unique bytecode. Therefore, native Java bytecode cannot be launched on Android systems directly. The Android community provides a tool (dx) that permits the substitution of Java class files into Dalvik executables (dex). The DVM implementation is optimized to improve efficiency and effectiveness on mobile devices that are configured with a slow (single) Processor, battery capacity, limited memory, and no swap space for the operating system. Also, the DVM was implemented such that it allows a device to execute VM's effectively. It depends on the revised Linux kernel for any possible threading and low-level memory features. DVM was replaced with ART (Android Runtime). ART was made official with Android 5.0 but candidly limited in its compatibility with all applications that are already on the market. ART introduces the concept of AOT (ahead-of-time) compilation, i.e., it compiles the whole application code into the native machine code, without interpreting the bytecode. Thus, enabling the application code to be executed directly by the device's runtime environment as compared to Dalvik's JIT (just-in-time) execution. AOT profiles the application while they are being executed and dynamically compiles the most used segments of the bytecode into native machine code.

Native C/C++ Libraries. Most main Android system infrastructures, such as DVM / ART and HAL, are native code-based, requiring native libraries compiled in C and C++. Android OS provides java framework APIs that display the capabilities of these core features to applications.

Libraries. The Library layer supports functionality such as 3D rendering using SGL (Scene Graph Library) and connects to databases using SQLite for Android to function effectively with its core features.

Application Framework. Developers can reuse and extend the components already presented in the API. This layer has managers, which enable Android applications to access data. These include; Activity manager, which controls the application's lifecycle and enables proper management of all the activities. Resource manager gives access to non-code resources. Notification manager allows the applications to show custom alerts in the status bar. Location manager notifies when a user enters or exits a specific geographical location. Package manager collects information about installed packages on the device. Window manager creates views and layouts. Telephony manager handles the structure of network connection and the information about services on the device. The Application Framework layer handles the API calls made by applications.

Applications. It is the most layer in android architecture that controls all installed apps on the device. For example, native applications include all pre-installed apps, such as camera, browser, SMS, calendars, contacts.

2.1.2 Android Application Components

The Android software development kit (SDK) is a set of tools needed to develop an Android app. Android apps are packed into an Android package (.apk) files through the Android Asset Packaging Tool (AAPT). AAPT has all the assets and compiled source code required to install an application on a device (Google, 2019b) based on Dalvik specifications. Figure 2.2 shows the Android app package structure. Google supplies the Android Development Tools (ADT) to streamline the development process. The ADT assembles the conversion from class to dex files and creates .apk during deployment.

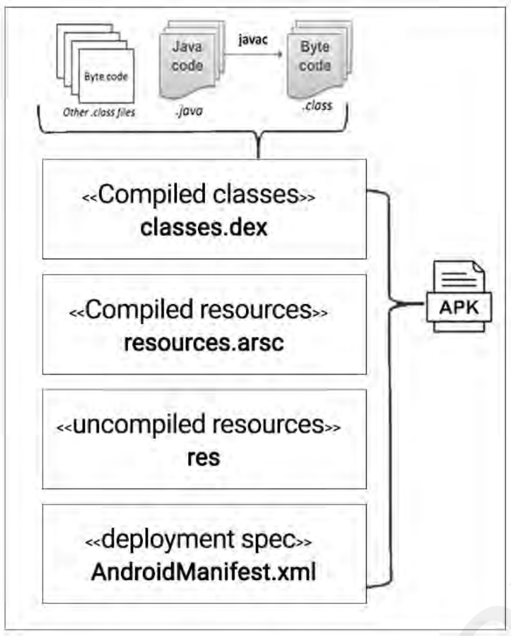


Figure 2.2: Android application package structure

Android manifest file. This is an indispensable XML (eXtensible Markup Language) file that is located at the root directory of the application's sources as AndroidManifest.xml. The manifest file is transformed into a binary format when the application is compiled. 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 needed, the list of permissions required, and the hardware specifications. In a very simplified manner, the four major components of an Android application are; Activities, Services, Broadcast Receivers, and Content Providers (Google, 2019b).

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, also known as widgets or controls (Google, 2019a). 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 user can switch between tasks by clicking the Home button and starting a new stack of activities on the mobile device. Activity manager is involved in the management of stacks and the activity lifecycle. Once the activity stops due to the launch of a new activity, a callback method is notified that allows the smooth transitioning of the activity. The layouts and widgets are described in the manifest file of Android apps, where each layout and widget have a unique identifier. Activity provides a platform for user's interaction with the application, creating a loop called windows, thus creating a space for the application to interact with the UI. This window is typically a full screen that floats on top of other windows but may be smaller than the screen. Each activity comprises a set of Views and Fragments that presents information to the user while interacting with the application. Fragments were introduced to address the issue of screen size and represent behaviors of a user interface in an activity.

Services are classes that do not provide the user with a screen for interaction as compared to Activity, and so can be executed in the background (for example playing music in the background; this is a long-running task). An app’s activities must be properly registered in the manifest for this component to function well. The three different types of services are (i) Scheduled which is characterized by jobs and requirements for network and timing. (ii) Started, it operates in the background indefinitely, even when the component that started it is no longer executing. (iii) Bound provides a client-server interface that permits the components to interact with the service. This is done by sending requests, receiving results, and processing with Inter-process Communication (IPC). All services must be declared in the application’s manifest file, just as activities and other components.

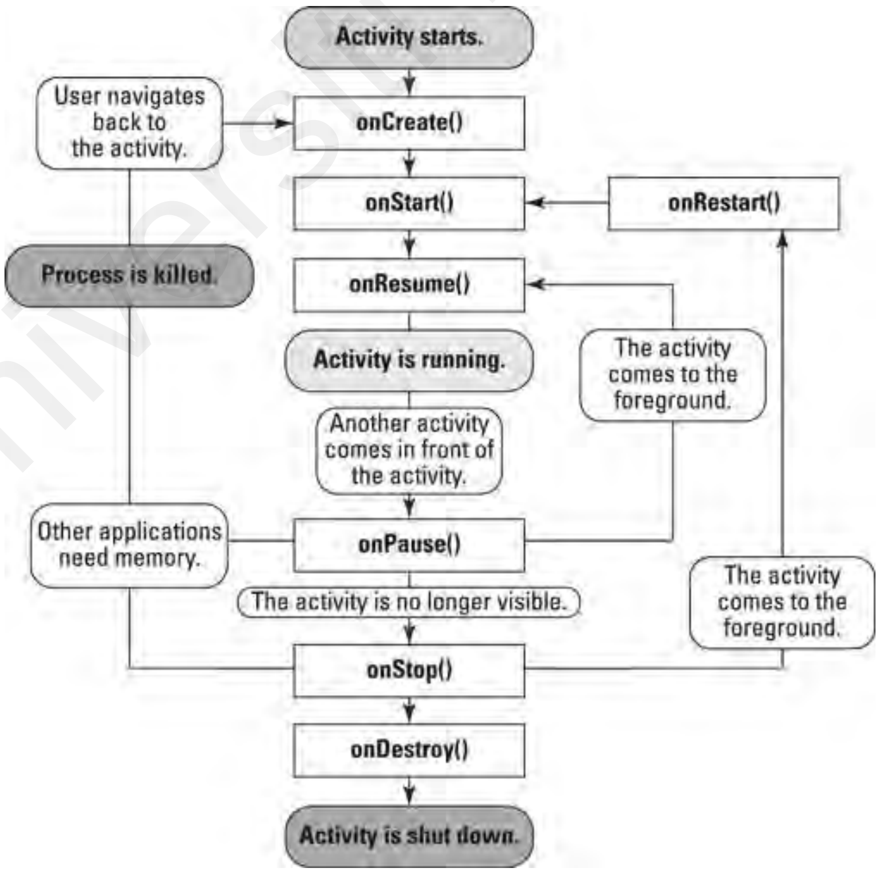
Broadcast Receiver is a component of an application that receives Intents from other applications that own the needed permissions. It listens and handles events related to particular states, either the system or other applications, similar to when a new message has been received or the OS has finished its initialization.

Content Provider oversees the structured data enclosed in the database. It permits the application to share information with other applications. Broadcast Receiver responds to messages from other applications or the system i.e., the application can initiate broadcasts to notify another app about the downloaded data to the device and its availability for use. This broadcast receiver intercepts the communication and will initiate appropriate action. Intent messages activate Services, Activities, and Broadcast Receivers. It has an inter-application message-passing framework. It is widely used in Android to transfer access from one activity to another, allowing late run-time binding between components, where the call codes are implicit and linked via the event messaging, an important feature of event-driven systems.

2.1.3 Android Activity Lifecycle

Activities on Android apps are components that are displayed for the users to interact with. They are usually composed of several loosely coupled activities bound to each other, and these activities can change the interface between different areas of application or perform different actions. For example, database accesses. Activities can call other activities using intents, which are used as links between the activities in the app. Intents are messages between the different components that are used to perform an activity (intention). The most common use for intents is starting a new activity and enabling the user to send extra data to the newly started activity, a bundle can be used, which acts as secondary storage when transferring data between activities.

An activity can have several different states, and the developer can override a corresponding method for each state in the source code i.e. if an activity has just been launched, the onCreate() method is invoked (Google, 2019k). This is called the activity lifecycle and can be the source of errors in Android apps. Figure 2.3 shows the Android activity lifecycle. The app's underlying process type is adjusted when the app components change their states. The basic individual unit begins when the app starts operation, while the following hooks are named sequential in activity: onCreate(), onStart(), onResume(). The first hook is called once in the lifetime of activity, but others get called more often. The onPause() method gets called when an activity loses its focus and when the activity is no longer visible, onStop() gets called. Also, before disabling an activity, the onDestroy() method continues to operate until the halt of the activity's lifetime. Every hook gets called specially, thus enabling the activity to maintain its state or restart correctly.



Activity starts.
User navigates back to the activity.
onCreate()
onStart()
onRestart()
Process is killed.
onResume()
Activity is running.
The activity comes to the foreground.
Another activity comes in front of the activity.
Other applications need memory.
onPause()
The activity is no longer visible.
The activity comes to the foreground.
onStop()
onDestroy()
Activity is shut down.

Figure 2.3: Android Activity Lifecycle

2.2 Graphical User Interface (GUI) Testing

The Graphical User Interface (GUI) is an app interface that provides a massive way for the user to communicate with the software in a modern software system and applies to more than half of the source code (Memon, 2003). The user interacts with software GUI by performing events such as click, long click, swipe, scroll, and adding text. Subsequently, the GUI engages with the runtime environment through message and callback methods. GUI is characterized into graphical orientation, event-driven input, the component they contain, and the attributes of those components.

GUI testing is a way to validate GUI components and the functionalities accessible through them. GUI testing is categorized into two namely usability testing which is not covered in this study, but mainly assesses how usable the interface is by using the tenets from user interface design and functional testing involves the assessment of an interface to test its workability, this is necessary to find out if the user interface works as intended (Ammann & Offutt, 2016). In general, GUI testing involves executing a task and comparing the outcome with the expected output. This is executed using test cases. GUI testing can be done either manually by humans and/or automatically. Memon published that GUI testing is manually operated (Memon, 2002). Additionally, testing GUI's can be difficult because the number of available GUI permutation actions are great, and each action may change the state of the program, and all of the action may need to be tested, and it is almost impossible to test all the states a graphical user interface can have (Kropp & Morales, 2010). GUI testing can be much work if performed manually, errors can be difficult to reproduce, and the process is infeasible (Kropp & Morales, 2010; Wang et al., 2014) while automated GUI testing is more accurate, reliable, efficient, and cost-effective than manual GUI testing (Li & Wu, 2004). GUI testing is not a single test activity; rather it is a collection of activities that test the app from various viewpoints, including test coverage, test case generation, test oracle, and regression testing. Test case generation is the major demanding task. Automated testing tools cannot substitute human intelligence for testing, but it would never be possible to test complex systems at a fair cost without them (Fewster & Graham, 1999).

Even though there is a variation between conventional software testing to GUI testing, both undergo similar testing steps that follow the pattern, respectively (Memon, 2001; Muccini et al., 2012). The first step is known as coverage criteria; these are guidelines used to assess what to test in software; this demands that each event be implemented and executed to determine its functionality and usability. The second step is the most important part of the test case, which is known as test input generation. Test inputs comprise events such as clicks, scroll, and object manipulations and are constructed based on software specifications or its structure. The third step is known as expected output generation; test oracles is a mechanism that helps generate the expected outputs which determine if the software was successfully executed or otherwise. The fourth step involves the execution of test cases and verification of outputs, here, test cases are executed, and the output generated is compared to its expected result.

A test case consists of an input, output, expected outcome, and the actual outcome. Lastly, this step determines if the GUI was properly and adequately tested. This step usually requires analyzing the software to check the parts that were tested and those that were missed during the testing exercise. Problems identified after testing are usually modified and corrected. This modification leads to regression testing, simply put re-testing the modified software.

Regression test shows the accuracy of the modified software component and assesses that the changes had not significantly impacted the previously tested parts.

2.2.1 GUI Testing on Android Application

Testing Android GUI application has special requirements compared to desktop GUI applications. Due to some factors; Android applications run on heterogeneous devices, where different manufacturers use different technologies, there are a number of bugs derived from the heterogeneous aspect (Amalfitano et al., 2011). Also, as compared to the monolithic and independent desktop application, the ubiquitous nature of the Android framework and its code execution pose difficulties (Mirzaei et al., 2015). Android applications can be customized on different types of devices by providing different layouts and functionality, depending on the device running the application. Moreover, the Android activity lifecycle (This includes testing the activities respond to the user, system, and its lifecycle events), Service testing, Content provider (testing shared resources). An example of such a shared resource is a database. Broadcast receiver tests the component listening to a message from an intent (Amalfitano, Fasolino, Tramontana, & Robbins, 2013). According to Google (Google, 2019f) stated that Android applications can have numerous entry points since the activities act as independent modules that are connected. This means that testing becomes more difficult since the transition between every activity needs to be tested as well (Zhauniarovich et al., 2015).

2.2.2 Faults in Android GUI Testing

Generally, Android application GUI testing aims to execute applications using a combination of inputs and states to reveal a fault. A fault is defined as a coding error in an application, also known as a Defect or Bug. Faults cause an application to crash during use, which may or may not lead to failure. Failure here means a system's inability to perform a necessary function within specified performance requirements (Maji et al., 2010). The source of failures includes faults in the application implementation, the running environment, and the interface between the application and its environment (Amalfitano, Fasolino, Tramontana, & Robbins, 2013).

An Android application crashes when there is an unexpected exit caused by an unhandled exception (Google, 2019d). Crashes typically result in termination of the application's processes, and a dialog is displayed to the user to notify them of the crash. Some crashes do not impact the execution visually, but the screen remains unresponsive. Once the crash is observed, the tester investigates the crash to find the fault that caused it and correct that fault. Android faults are categorized as activity errors and event errors (Hu & Neamtiu, 2011). Activities are the major GUI features of an Android application; an activity error happens due to inaccurate implementation of the Activity class. Event errors arise when an application refuses to respond as a result of getting an event. By configuration, Android applications are expected to be ready to receive and respond to events at any stage of activity in which they occur, e.g., an application must be able to manage the intrusion caused by an incoming phone call in any state. If developers do not provide the effective implementation of event handlers associated with certain states, the application may enter an erroneous state or crash as a result of an event.

2.2.3 GUI Testing Frameworks for Android

There are varied GUI testing tools and frameworks for Android applications include Android Monkey (Google, 2019j), Appium (Sauce, 2013), Espresso (Google, 2019e), Robotium (Reda & Josefson, 2014), UiAutomator (Google, 2019i), and Monkeyrunner (Google, 2019g). These frameworks are not limited to the stated frameworks but were identified based on their popularity in the software marketplace (Gunasekaran & Bargavi, 2015). Automated GUI testing frameworks have been used to compile and execute test cases for Android applications and make the testing activity easier.

Android Monkey (Google, 2019j), 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. Also, the generated events are low level with hard-coded coordinates, which complicates the reproduction and debugging processes (Choudhary et al., 2015). Moreover, Android Monkey is unable to turn the sequence of events into test cases.

Appium (Sauce, 2013) is an open-source black box testing, cross-platform mobile app automation testing framework developed by Sauce Labs to automate native, hybrid, and mobile web applications. It is OS independent, but not a device-independent that uses UiAutomator or Selendroid for running the tests in the background. Appium interacts with multiple applications once the Android API level on the test device is greater than what is required by UiAutomator. Appium is best known for its accuracy during automation testing and test repeats and also can be developed on any language because it is not directly paired to Android. Appium has limited support for hybrid testing and cannot be used on Android version lower 4.2.

Espresso (Google, 2019e) is a user interface-testing framework for testing android app developed in Java / Kotlin language using Android SDK. It is an open-source project used majorly to write a functional UI test whose focus is on the navigation of AUT. Espresso is highly reliant on the instrumentation framework of Android, and therefore, limited by its inability to navigate outside the AUT. The test is simple to compile, since Espresso ensures that the application is at a stable before proceeding with the test script. Stable state means that the application is not waiting for an animation or network call to end (Nolan, 2015). Once the intent has been stated, Espresso takes care of timings. This tool requires access to the application's sorce code which may inhibit the process of testing and it has a narrow focus such that test are written twice for two different systems if UI tests are needed for both Android and iOS.

UiAutomator (Google, 2019i) is a UI testing framework produced by Google as part of Android SDK. It uses for cross-app functional testing on apps and across the platform. This is similar to Espresso, because it comes with Testing Support Library included as a part of the test package but it does not use the Instrumentation framework. It is a black box testing that needs to be in a highly stable before the continuation of test execution as compared to Espresso.

Robotium (Reda & Josefson, 2014) is an open-source project, by Renas Reda. It was designed to abstract the structure of instrumentation that is hard to use. The test development language is Java. Similarly to the previously mentioned tools, Espresso and UiAutomator, Robotium have access to the classes and methods publicly available in the AUT since the test code and AUT's source code are included in the same project ("Robotium" 2016). Like Espresso, as it is based on the instrumentation system, Robotium is restricted to traversing within the AUT. The tool is in active development. In contrast, Robotium is a third party tool and so requires the library is simply loaded to the Android app project.

Monkeyrunner (Google, 2019g) is used for GUI testing. The language used here is Jython, a Python implementation that uses Java. Google designed it and includes Android SDK. This tool sends inputs, such as touch events or key strokes, to the AUT and takes screen shots of the application as compared to other tests. It possesses the capability to input test suites into numerous emulated devices and boot up said emulators.

Monkeyrunner is effective in automating the execution of the test suite. Monkeyrunner works on all Android versions, and functions on a level lower than the Android Framework. Compared to other tools, it lacks high-level methods for searching and asserting items.

Most of the above solutions focus on automating manual efforts in the GUI testing process instead of improving the test efficiency of the complex modern application GUI (Septian & Alianto, 2018). Frameworks such as Robotium, UiAutomator, and MonkeyRunner provide a set of APIs for the tester to write test scripts based on their test requirements. They do not provide a way for generating the test cases automatically; hence the test cases have to be developed manually. Moreover, writing the test cases using these frameworks and scripts for all the available and upcoming applications is impractical and non-effective in terms of time, effort, and cost. In contrast, the main focus of this dissertation is on implementing an approach for generating test cases automatically.

2.3 State-of-the-Art GUI Testing Tools for Android

This section highlights and discusses a thematic taxonomy for the classification of Android application GUI testing tools. The taxonomy was created by analyzing 45 different studies from 2011 to 2019. The studies were from five known data sources: springer, IEEE Xplore, ACM, science direct, and google scholar.

2.3.1 Taxonomy of GUI Testing Tools for Android

The taxonomy contains four prominent parameters, which are (i) automated testing activities, (ii) GUI testing tools approach, (iii) evaluation methods, (iv) and characteristics, as illustrated in Figure 2.4.

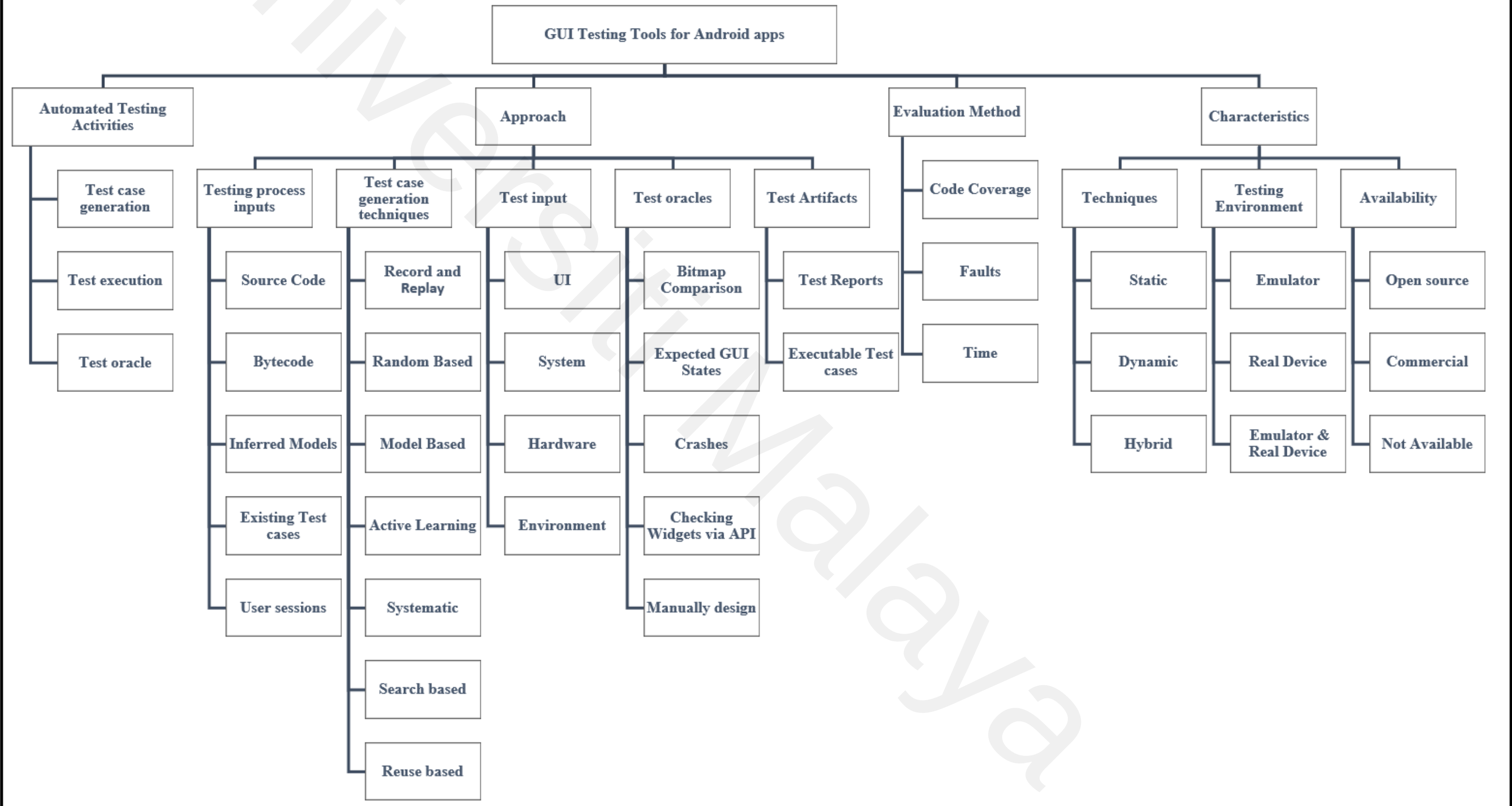


Figure 2.4: Taxonomy of GUI testing tools for Android Applications

2.3.1.1 Automated Testing Activity

Automation is a key factor for any Android application testing. Automated testing includes test case implementation and design, test execution, and test oracle definition and evaluation that verify the test by comparing the output results with the expected results (Shahamiri et al., 2009).

2.3.1.2 Approach

GUI testing tools applied different techniques to design and implement test cases and test oracle by using different inputs such as source code, bytecode, inferred model, existing test cases, and user session.

Testing process input is the source of information required by the tool to derive test cases such as (1) Source code when the tool requires source code of the Android application; (2) Bytecode when the tool requires the executable code to derive test cases; (3) Inferred model uses AUT model that generates automatically or manually to derive test cases; (4) User session uses record and replay techniques to record and re-execute user sessions that can be transformed to executable test cases; (5) Existing test cases have been created by changing existing sessions, often in the form of executable Junit test cases for Android applications, into executable test cases.

Test case generation is one of the most attention-demanding testing activities because of its strong impact on the overall testing process efficiency (Anand et al., 2012). The total cost, time, and effort required for the overall testing will depend on the total number of test cases. The effort depends on the size of the application and the number of test cases. The test case comprises input, output, expected result, and actual results. A set of test cases is referred to as a suite. The test suite provides detailed guidelines or goals for every test case collection (Memon, 2019). Test generation techniques include:

(1) Record and replay, which records user interactions with GUI components into a test script that can replay automatically on AUT to mimic user usage. Test cases are generated and executed based on the scenarios recorded. Also, test data from the scenarios can be modified as per requirement during the execution.

(2) Random is based on probability and distinct events generated by the GUI application to trigger faults and crashes (Liu et al., 2010). The set of all GUI actions available in the current state is identified, yet another GUI action is chosen from the set and sent to the AUT for execution.

(3) Model-based uses a graph-based model to represent the user interaction with the app's GUI. The model is designed either manually or automatically by adopting the AUT's specifications, such as code or XML configuration files, or through direct interaction with the apps. Test cases are produced based on a model abstraction according to specific test selection criteria, such as coverage (Utting et al., 2012).

(4) Active Learning is a combination of GUI testing and model learning technique (Amalfitano et al., 2015a). Developers will generate user event sequences based on the model of the Android application GUI. This technique may exploit the known graph exploration algorithms like Breadth-first search (BFS) or Depth-first search (DFS), or a combination of both, to test the Android application GUI.

(5) Systematic approach uses more sophisticated techniques such as program analysis. 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. Symbolic execution was introduced by (King, 1975), which is a program analysis technique where symbolic values are used as program inputs instead of concrete values. Then the output of the program is transformed into a function of the symbolic inputs. Symbolic execution is

computationally expensive, and it is difficult to reason about all paths of a significantly large program symbolically. Therefore, concolic execution has been proposed to alleviate the path explosion problem. Concolic execution is known as dynamic symbolic execution, is a combination of symbolic and concrete execution techniques.

(6) Search-based uses meta-heuristic search algorithms (Saeed et al., 2017) such as genetic algorithms (GA) for testing. This technique generates test cases driven by an objective function that is specified according to a desirable test goal. Evolutionary algorithms (Zitzler & Thiele, 1999) have been used to test programs (Sharma et al., 2014). However, the limitation of the evolutionary algorithm-based approach that maximizes test coverage is that it takes a long time to produce a test suite.

(7) Reuse based techniques rely on existing test cases manually written or automatically generated for deriving new tests that may be executed in various background conditions.

Test input is the data used to execute test cases, which include UI user events, system events, hardware events, and external events. UI events are user interactions with the system through the user interface, such as touch, text, and scrolls inputs. User interactions over the touchscreen are the primary source of input for Android applications. Android describes UI events using MotionEvent and KeyEvent classes, each extending the InputEvent class. MotionEvent specifies the user input in terms of an action code (e.g., ACTION_Up, ACTION_DOWN) and screen coordinates. A sequence of MotionEvents can describe any user gesture such as long-press, fling, and pinch. KeyEvents describe a key that has been pressed (e.g., volume, virtual keyboard). Android framework provides a variety of GUI elements, also known as widgets or controls. These widgets include TextViews, ImageViews, or ScrollViews which allow the users to interact with the apps.

System events are a messaging system across apps and outside of the regular user flow, such as receiving SMS notification, application notification, or 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. Hardware events are generated from the device hardware like a battery or peripheral port such as USB, headphone, network receiver/sender. External events are generated by the external environment and sensed by device sensors such as temperature, pressure, GPS, and geomagnetic field sensor. Many methods to identify event inputs such as Dumpsys, event patterns, and permission-based. 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, 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 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.

Test oracle is considered a challenging activity of the testing process to be automated (Barr et al., 2014; Shahamiri et al., 2009). The following is a brief explanation of the techniques used for the validation: (1) Bitmap Comparison: the process of verifying test execution by comparing visual object state using screenshots and images of widgets, with the expected state of the object. (2) Expected GUI States: when states of the GUI extracted during the first execution of the AUT are used to verify further execution of test cases.

(3) Widgets via API technique: capture widgets of the Android application GUI and compares them with the widgets captured with the help of API's on the same GUI to verify the correctness (Amalfitano, Fasolino, Tramontana, & Robbins, 2013). (4) Crash detection or exceptions: the distinctive implicit way to assess the results of the test case. Test case is marked as failed if the AUT crashes during execution else it is considered a pass. (5) Manually design oracle: a process of verifying the output results of test execution manually.

Test Artifacts are the outputs generated by the tool during testing, which include test reports such as coverage reports or testing output and executable test cases that are executed outside the context of the generated process, for example, Junit test case.

2.3.1.3 Evaluation Methods

Evaluation methods are criteria used to evaluate the success and performance of the tool, such as (1) Coverage is an important metric to measure the effectiveness of testing (Memon et al., 2001). Coverage criteria include code coverage and activity coverage. Code coverage evaluates statement coverage, line coverage, block coverage, instruction coverage, branch coverage, and method coverage. It provides a measurement of how much source code was executed during the tests. Activity coverage is defined as the ratio of activities explored during the execution to the total number of activities present in the application. Activities provide the main interfaces for interaction with the end-user. (2) A fault is the capability of the tool to detect faults such as application crashes during real-time execution. 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. (3) Time criteria are used to evaluate the tool by calculating the duration of the test process.

2.3.1.4 Characteristics

The characteristics of the GUI testing tools in terms of the techniques to generate test cases (i.e., static analysis, dynamic analysis, or hybrid), testing environment (i.e., emulator and real device), source code availability, and tools basis.

Techniques applied in the test case generation process are used to analyze the Android application to identify GUI components and event-driven behavior. 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. Static analysis techniques generate tests based on data such as source code or high-level models of AUT. It analyzes the Android app's events handler statically and triggers events on the basis of AUT's source code without executing the app. The static analysis reviews and detects faults that 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. The hybrid analysis combines dynamic and static analysis.

Testing Environment is a platform that supports hardware and software test execution. The test environment is typically designed according to the specifications of the application under test. There are two main options for Android apps to conduct GUI tests: (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 users' experience on the app. Real devices produce real results and live network performance defects. An emulator is a virtual mobile device that runs on a computer to mimic some hardware and software features of a real device. It can be used to test the app against the 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 a part of the Android software development kit (SDK). However, the emulator is not able to emulate events like battery issues, network connectivity, and gestures.

Availability in this context refers to the availability of the tool's source code to be accessed by the public.

2.3.2 Comparison of GUI Testing Tools for Android

This section addresses the overview of the current state-of-the-art GUI testing tools for Android applications. Tables 2.1, Table 2.2, Table 2.3, and Table 2.4 classify GUI testing tools based on categories of the thematic taxonomy presented in section 2.3.1.

2.3.2.1 Automated Testing Activities

GUI testing is classified into three main activities, test case generation, test execution, and test oracle definition and evaluation. As illustrated in Table 2.1, most of the literature's tools focus on test case generation and execution but do not support test oracle. An automated test generation technique alone is not sufficient because a human tester must manually determine whether each test case diverges from the expectation. An automated test oracle is a technique that will help to mitigate this problem. To implement an automated test oracle, you will have to export test cases to executable test scripts. Immediately the test case is materialized as a script. Moreover, a human tester can implement a test oracle. This is done by adding assertions to the script.

Table 2.1: Comparison of Automated testing activities

Attribute	GUI Testing Tools
Fully Automated **Test case generation, Test execution, and Test oracle definition and evaluation**	Droidbot (Li et al., 2017), AimDroid (Gu et al., 2017), Stoat (Su et al., 2017), Barista (Fazzini et al., 2017), Sapienz (K. Mao et al., 2016), Crashscope (Moran et al., 2016), Thor (Adamsen et al., 2015), Cadage (Zhu et al., 2015), MobiGuitar (Amalfitano et al., 2014), SlumDroid (Imparato, 2015), AppDoctor (Hu et al., 2014), PUMA (Hao et al., 2014), ACRT (Liu et al., 2014), Dynodroid (Machiry et al., 2013), AndroidRipper (Amalfitano, Fasolino, Tramontana, De Carmine, & Memon, 2012), Tema (Takala et al., 2011).
Test Case Generation and Execution	APE (Gu et al., 2019), Amoga (Salihu et al., 2019), SmartMonkey (Haoyin, 2017), (Zeng et al., 2016), TrimDroid (Mirzaei et al., 2016), FSMdroid (Su, 2016), T+ (Linares-Vásquez, 2015), MonkeyLab (Linares-Vásquez et al., 2015), PATS (Wen et al., 2015), Sig-Droid (Mirzaei et al., 2015), EvoDroid (Mahmood et al., 2014), DroidCrawle (Wang et al., 2014), ORBIT (Yang et al., 2013), A3E (Azim & Neamtiu, 2013), RERAN (Gomez et al., 2013), SwiftHand (Choi et al., 2013), Collider (Jensen et al., 2013), Extended Ripper (Amalfitano, Fasolino, Tramontana, & Amatucci, 2013), ACTEve (Anand et al., 2012), GUI Ripper (Amalfitano, Fasolino, Tramontana, De Carmine, & Imparato, 2012), Testdroid (Kaasila et al., 2012), A2T2 (Amalfitano et al., 2011), (Hu & Neamtiu, 2011).
Test Case Generation	(Zheng et al., 2017).
Test Execution and Oracle definition and evaluation	GUICC (Baek & Bae, 2016), iMPAcT (Morgado & Paiva, 2015), SPAG-C (Lin et al., 2014).

2.3.2.2 Approach

Testing process inputs. The studied tools exploit one or two source information to generate a test case. White box approach applied in thirteen tools is reliant on the analysis of the source code of AUT. Black-box approach requires the executable code that is applied in nineteen tools. Grey box approach uses source code and executed code, which is applied in six tools. The inferred model is analyzed to derive test cases. Models are designed manually, FSMdroid (Su, 2016) design finite state machines for Android application under test. DroidCrawle (Wang et al., 2014) and PATS (Wen et al., 2015) automatically design GUI trees by reverse engineering process. User sessions transform into executable test cases by using record and replay techniques.

Test case generation is a key feature for most literature tools. Several techniques have been used for generating test cases, namely Model-based technique, Active learning technique, Random based technique, Search-based technique, Systematic based, and Record and Replay. However, the most popular technique is the Model-based technique. This technique is dependent on two models, the high-level models, and low-level models. The high level is composed of behavioral models such as finite state machines, sequence diagrams, activity diagrams, event flow graphs, and GUI trees, and low-level models that relate directly to the AUT code, such as control flow graphs or call graphs. The model is used in GUI testing to guide the exploitation of an application, circumnavigate the model which uses specific instructions to generate action sequences systematically, and then playback action sequences of the test application. GUI testing tools adopted active learning techniques; the models generate test cases automatically at runtime during the testing process. Although other existing test cases such as reuse-based, for example, Thor and Extendedripper have infused specific sequences of events in existing replicate test cases.

The existing test cases test the AUT robustness with respect to system events, i.e., sending an intent to start and restart the app or on/off WiFi. Random-based techniques use uniform random techniques, for example, Dynodroid and AppDoctor, and smarter random techniques such as SmartMonkey for test case generation. Finally, search-based testing techniques rely on using an evolutionary algorithm.

Test input is the main activity to generate test cases. Generally, all tools generate test inputs but vary. While all the tools generate UI event inputs, some generate system events and few text inputs. Text inputs can be generated automatically or manually, and the input types are concrete, predefined, contextual, and random. Pieces of literature, such as Droidbot, fill in text input fields by searching for a sequence of predefined inputs. When none of the predefined inputs can satisfy the input restraints, these predefined inputs may fail to exercise beyond the input. Sapienz, Stoat, and Android Monkey produced random text input. Dynodroid paused the test for manual input when encountering a text input field, such as the login password. System events can effectively expose app faults from context events. Android apps are context-aware because of their ability to sense and react with a great number of different events that come from the user or system interactions. However, most of the recent testing tools for Android apps focus on UI events. Thus, they make it difficult to identify other defects in the changes that can be preferred by the context in which an app runs.

Test Oracles determine the executed test case result. Test oracle has been proposed in seventeen of the studied GUI testing tools. In eleven tools, crash detection represents the unique implicit way to evaluate the result of executed test cases. Crashscope (Moran et al., 2016), MobiGUITAR (Amalfitano et al., 2014), Dynodroid (Machiry et al., 2013), TestDroid (Kaasila et al., 2012), and (Hu & Neamtiu, 2011) manually verify test cases by analyzing execution log and/or crash reports. Moreover, Barista (Fazzini et al., 2017), EHBDroid (Song et al., 2017), ACRT (Liu et al., 2014), and Collider (Jensen et al., 2013) inspect crashes and confirm their validity manually. Manual oracle is resource-intensive because the testers interact with the GUI manually to generate events and visually detect the error. Bitmap comparison techniques compare the actual state screenshot with the expected state image used in SPAG-C (Lin et al., 2014) that snapped a photo of the actual state with an external camera. The studies checked the value or the condition of the widget using the Android API, A2T2 (Amalfitano et al., 2011), and AppDoctor (Hu et al., 2014) by verifying states or values of a widget using Android API. GUI verification through an API has a high degree of maintainability. Moreover, APIs sometimes are not available, and API access is that the gathered data from the API might be slightly different when seen through the GUI. Droidbot (Li et al., 2017) assesses the invariants obtained from the assessment of common application bugs.

Test Artifacts are obtained during the test case generation process. The test artifacts generated are grouped into the following categories: test execution outputs such as test reports, test coverage, and executable test cases. The most common type of test artifact is the executable test case. These tests can only be tested by the same tools that generate them. However, few tools can traverse the generated test cases, allowing them to be executed outside the tool of the test generation process. Test execution can also be presented as output, which provides reports of crashes, logs, and code coverage. For example, Dynodroid replicates test case reports when the tool crashes, Android Ripper creates executable JUnit test cases, RERAN replicates the same user sessions that have been captured, and Sapienz generates event sequences for the easy derivation of the test case, such as crash and code coverage.

Table 2.2: Comparison of GUI test case generation tools

Tools	Testing Process Inputs	Test Case Generation Techniques	Test Inputs	Test oracle	Test Artifacts
APE	Bytecode Executable Automatically Inferred Model	Model-based	User events	-	Test Reports (crash & coverage)
Amoga	Bytecode Executable, Automatically Inferred Model	Model-based	User events	-	Executable test cases
AutoDroid	Bytecode Executable,	Random	User events	-	-
AimDroid	Bytecode Executable	Model-based	User events	-	
DroidBot	Bytecode Executable, Automatically Inferred Model	Active learning	User and System	Invariant Conditions	Test Reports Activity coverage and log)
SmartMonkey	Bytecode Executable	Random	User and System	-	-
EHBDroid	Source code		User and System	-	-
Barista	User session	Record and Replay	User events	Manually design	Executable test cases
Stoat	Source code, Bytecode execution	Model-based	User and System	-	Test reports (crash, coverage & log), Executable test cases
(Zheng et al., 2017)	Source code	Random	User events	-	Coverage report
Sapienz	Source code, Bytecode execution	Search based	User and System	Crashes / Exceptions	Executable test cases, (crash, coverage & log)
(Zeng et al., 2016)	Source code	Model-based	User events	-	Test output, Executable test cases
Crashscope	Bytecode Executable	Model Learning	User and System	Crashes	Test output, Executable test cases

Table 2.2: Continued

Tools	Testing Process Inputs	Test Case Generation Techniques	Test Inputs	Test oracle	Test Artifacts
TrimDroid	Source code, Automatically Inferred Model	Model-based	User events	-	Test output, Executable test cases
FSMdroid	Source code, Manually Inferred Model	Model-based	User events	-	Coverage report
Thor	Existing test case	Reuse Test case	User and System	Crashes	Test output, Executable test cases
T+	Bytecode Executable, User session	Model-based, User session	User events	-	Test output, Executable test cases
Cadage	Bytecode Executable, Automatically Inferred Model	Model Learning	User events	Crashes	-
MobiGuitar	Source code	Model-based	User events	Crashes	Test output, Executable test cases
MonkeyLab	Bytecode Executable, user session	Model-based, User session based	User events	-	Test output, Executable test cases
PATS	Bytecode Executable	Model-based	User events	-	Test output, Executable test cases
Sig-Droid	Source code, Automatically Inferred Model	Systematic	User events	-	Test output, Executable test cases
SlumDroid	Source code	Model learning	User events	Crashes	Test output, Executable test cases
EvoDroid	Source code, Automatically Inferred Model	Search-based	User events	-	Coverage and crash report
SPAG-C	User session	Record and Replay	User events	Expected Bitmap	-
AppDoctore	Bytecode Executable, Existing Test cases	Random testing	User events	Expected GUI state. Crashes	Test report (log), Executable test case
DroidCrawle	Source code	Model Learning	User events	-	Coverage report
PUMA	Bytecode Executable	Model-based	User events	Expected GUI state, Crashes	Test report (log)
ACRT	Bytecode Executable, user session	Record and Replay	User events	Manually Designed	Test outputs, Executable test case

Table 2.2: Continued

Tools	Testing Process Inputs	Test Case Generation Techniques	Test Inputs	Test oracle	Test Artifacts
Dynodroid	Source code	Random	User and System	Crashes	Test report (Coverage & Crash report)
ORBIT	Source code, Automatically Inferred Model	Model-based	User events	-	-
A3E	Bytecode Executable, Automatically & Manually Inferred Model	Model-based, Active learning	User and System	-	Test output, executable test cases
RERAN	Bytecode Executable, user events	Record and replay, User session based	User and System	-	Test output, executable test cases
SwiftHand	Bytecode Executable	Active learning	User events	-	Test output, executable test cases
Collider	Manually Inferred Model	Systematic	User events	-	Branch coverage
Extended Ripper	Bytecode Executable, Source code	Reuse Test case	User and System	-	Test output, executable test cases
Android Ripper	Bytecode Executable, Source code	Model-based	User events	Crashes	Test output, executable test cases
ACTEve	Source code	Systematic	User and System	-	Test output, executable test cases
GUI Ripper	Bytecode Executable, Source code	Model-based	User events	Crashes	Test output, executable test cases
Troyd	Bytecode Executable	Record and Replay	User events	-	-
Testdroid	Bytecode Executable, Existing Test cases	User session based	User events	-	Test report (log)
A2T2	Bytecode Executable	Model-based	User events	-	-
(Hu & Neamtiu, 2011)	Source code	Random	User events	Crashes	Test reports (log)
TEMA	Bytecode Executable	Model-based	User events	Crashes	Test reports (Crash report)
PUMA	Bytecode Executable	Model-based	User events	Expected GUI state, Crashes	Test report (log)

2.3.2.3 Evaluation Methods of GUI Testing Tools for Android

Evaluation methods are used to measure the effectiveness of tools. The most frequently considered effectiveness metric is the number of faults that have been found (i.e., crashes or exceptions). Besides, coverage criteria (such as line coverage, statement coverage, branch coverage, method coverage, block coverage, Instruction coverage, and activity coverage) have been measured. Most of the tools combine different granularities of coverage metrics that can be beneficial for achieving better results in testing the Android app. Table 2.3 illustrates the evaluation methods of GUI testing tools.

2.3.2.4 Characteristics of GUI Testing Tools for Android

The GUI testing tools have several features that ensure the successful evaluation of the Android application. Table 2.4 reports a summary of the characteristics of the GUI testing tools for Android applications.

Techniques have been categorized by differentiating between static and dynamic analysis techniques. Many of the tools have been based on static analysis, while others use dynamic analysis, as shown in Table 2.4. Dynamic analysis can provide additional information that is not available statically, for example, whether certain UI widgets are disabled in a particular state. The static analysis exposes behaviors that are only possible under complex run time conditions that are unlikely to be triggered by automated dynamic exploration. The most natural direction to pursue is a hybrid static and dynamic analysis. Existing works by Amoga (Salihu et al., 2019), Stoat (Su et al., 2017), Sapienz (K. Mao et al., 2016), CrashScope (Moran et al., 2016), TrimDroid (Mirzaei et al., 2016), FSMdroid (Su, 2016), MonkeyLab (Linares-Vásquez et al., 2015), EvoDroid (Mahmood et al., 2014), AppDoctor (Hu et al., 2014), A3E (Azim & Neamtiu, 2013), ACTEve (Anand et al., 2012) have already considered this possibility. With the information inferred by static analysis (e.g., the events supported by a UI widget, or the possible GUI transition related to a widget), the dynamic analysis can be made more efficient and complete.

Table 2.3: Type of evaluation method of GUI testing tools

Evaluation Method	GUI Testing Tools
Line Coverage	Stoat (Su et al., 2017), TrimDroid (Mirzaei et al., 2016), MobiGuitar (Amalfitano et al., 2014), SlumDroid (Imparato, 2015), Sig-Droid (Mirzaei et al., 2015), EvoDroid (Mahmood et al., 2014), Dynodroid (Machiry et al., 2013), Extended Ripper (Amalfitano, Fasolino, Tramontana, & Amatucci, 2013), AndroidRipper GUI Ripper (Amalfitano, Fasolino, Tramontana, De Carmine, & Memon, 2012).
Statement coverage	Amoga (Salihu et al., 2019), Sapienz (K. Mao et al., 2016), Crashscope (Moran et al., 2016), TrimDroid (Mirzaei et al., 2016), FSMdroid (Su, 2016), MonkeyLab (Linares-Vásquez et al., 2015), ORBIT (Yang et al., 2013).
Block Coverage	Cadage (Zhu et al., 2015)
Method coverage	APE (Gu et al., 2019), AimDroid (Gu et al., 2017), Sapienz (K. Mao et al., 2016), A3E (Azim & Neamtiu, 2013)
Instruction coverage	APE (Gu et al., 2019), AimDroid (Gu et al., 2017)
Branch coverage	SwiftHand (Choi et al., 2013), Collider (Jensen et al., 2013), ACTEve (Anand et al., 2012).
Activity coverage	APE (Gu et al., 2019), AimDroid (Gu et al., 2017), Sapienz (K. Mao et al., 2016), AppDoctor (Hu et al., 2014), DroidCrawle (Wang et al., 2014), A3E (Azim & Neamtiu, 2013).
Faults	APE (Gu et al., 2019), Droidbot (Li et al., 2017), AimDroid (Gu et al., 2017), SmartMonkey (Haoyin, 2017), Stoat (Su et al., 2017), Barista (Fazzini et al., 2017), Sapienz (K. Mao et al., 2016), Crashscope (Moran et al., 2016), Thor (Adamsen et al., 2015), Cadage (Zhu et al., 2015), SlumDroid (Imparato, 2015), MobiGuitar (Amalfitano et al., 2014), SPAG-C (Lin et al., 2014), AppDoctor (Hu et al., 2014), PUMA (Hao et al., 2014), Dynodroid (Machiry et al., 2013), RERAN (Gomez et al., 2013), AndroidRipper (Amalfitano, Fasolino, Tramontana, De Carmine, & Memon, 2012), GUI Ripper (Amalfitano, Fasolino, Tramontana, De Carmine, & Imparato, 2012), (Hu & Neamtiu, 2011), Tema (Takala et al., 2011)
Time	Amoga (Salihu et al., 2019), SmartMonkey (Haoyin, 2017), Sig-Droid (Mirzaei et al., 2015), EvoDroid (Mahmood et al., 2014), AppDoctor (Hu et al., 2014), ORBIT (Yang et al., 2013).

Testing Environment. To execute the test case, the studies run the test on the emulator increase the risk of missing important bugs that occur only on real devices. Emulators are not the same as a real devices, which is a tremendous disadvantage. However, they offer diversity in terms of the different devices, operating systems, and adaptations of the user interface. Other tools support both real devices and emulators to consider heterogeneous devices, which may enhance the reliability of the testing. Moreover, testing activity uses cloud services infrastructure like TestDroid (Kaasila et al., 2012) and AppDoctor (Hu et al., 2014). TestDroid operates as a mobile devices cluster connected to the internet that allows users to upload their AUT to the system; thus, receiving the results from their system's web page account.

Availability in this context refers to the availability of the tool's source code to be publicly accessible. The source code of twenty GUI testing tools is available in the context of GitHub projects. In contrast, three of these tools are available only in executable form or in the demo version (i.e., Barista, TrimDroid, and AndroidRipper). Moreover, Sapienz source code is available on GitHub. However, it is outdated and not supported. At the same time, other tools like APE, Stoat, Droidbot, AimDroid, Thor, SlumDroid, AppDoctor, PUMA, Sig-Droid, Dynodroid, RERAN, SwiftHand, ACTEve, and GUI Ripper source code are easily accessible. Contrarily to the above tools, Android Monkey is available with Android SDK. Crashscope source code is not available, but the tool is obtainable on the internet. Also, the available version of ExtendedRipper only supports the Windows operating system. A3E is partially available.

BASIS is the underlying tool to design and build new tools. GUI testing tools usually rely on existing tools and/or libraries. The commonly used library is the Robotium (Reda & Josefson, 2014) library, which supports the compilation of Junit test cases. Other similar libraries are UIAutmator. Hierarchyviewer and Emma. The Emma library can be easily accessed in the Android framework. It is used to measure code coverage. Other relevant tools provided by the Android framework are Android Monkey and MonkeyRunner. Tools such as SmartMonkey (Haoyin, 2017) were built on top of Android Monkey and such 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 APE (Gu et al., 2019), Sapienz (K. Mao et al., 2016), (Zheng et al., 2017) and (Hu & Neamtiu, 2011). Dynodroid (Machiry et al., 2013) is based on MonkeyRunner. Other tools Sig-droid (Mirzaei et al., 2015), EvoDroid (Mahmood et al., 2014), and Collider (Jensen et al., 2013) are based on Java PathFinder. The JavaPath Finder performs a symbolic analysis of the java source code of Android applications. GUIRipper served as a base tool for ExtendedRipper (Amalfitano, Fasolino, Tramontana, & Amatucci, 2013), 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. Besides, Stoat (Su et al., 2017) is an upgraded version of the A3E (Azim & Neamtiu, 2013), and is used due to its unavailability to the public repository. However, Stoat has an enhanced UI exploration strategy and static analysis.

Table 2.4: Characteristics of GUI testing tools

Attribute		GUI Testing Tools
Techniques	**Static Analysis**	(Zheng et al., 2017), Thor, Sig-droid, Collider
	Dynamic Analysis	APE, DroidBot, AimDroid, SmartMonkey, Barista, (Zeng et al., 2016), GUICC, iMPAcT, T+, Cadage, MobiGuitar, PATS, SlumDroid, SPAG-C, DroidCrawle, PUMA, ACRT, Dynodroid, ORBIT, RERAN, SwiftHand, Extended Ripper, AndroidRipper, GUI Ripper, Testdroid, A2T2, Hu et al., Tema
	Hybrid	Amoga, Stoat (Su et al., 2017), Sapienz (K. Mao et al., 2016), CrashScope, TrimDroid, FSMdroid, MonkeyLab, EvoDroid (Mahmood et al., 2014), AppDoctore, A3E (Azim & Neamtiu, 2013), ACTEve (Anand et al., 2012)
Testing Environment	**Emulator**	Amoga, Sapienz (K. Mao et al., 2016), TrimDroid, FSMdroid, Thor, T+, Cadage, MobiGuitar, MonkeyLab, PATS, Sig-Droid, SlumDroid, EvoDroid, DroidCrawle, ACRT, Dynodroid, SwiftHand, Collider, ExtendedRipper (Amalfitano, Fasolino, Tramontana, & Amatucci, 2013), AndroidRipper, ACTEve, GUI Ripper, A2T2, (Hu & Neamtiu, 2011), Tema
	Real Device	(Zheng et al., 2017), Barista, SPAG-C, AppDoctore, RERAN, Testdroid (Kaasila et al., 2012)
	Emulator and Real device	APE, DroidBot, AimDroid, Stoat, CrashScope, (Zeng et al., 2016), GUICC, iMPAcT, PUMA, ORBIT, A3E
Availability	**Open-source**	APE, Barista, DroidBot, AimDroid, Stoat, Sapienz, TrimDroid, Thor, SlumDroid, AppDoctore, PUMA, Sig-Droid, Dynodroid, A3E-Dynamic, RERAN, SwiftHand, ExtendedRipper, AndroidRipper, ACTEve, GUI Ripper
	Commercial	Testdroid (Kaasila et al., 2012)

2.4 Comparison of Test Case Generation Approaches for Android

This section compares the state-of-the-art GUI test generation approaches and corresponding tools for Android applications.

2.4.1 Record and Replay

In this technique, user interactions with AUT components are recorded and converted into a test script, which is replayed automatically. User interaction is captured either on a GUI component level, e.g., via direct references to the GUI components, or a GUI bitmap level, with coordinates to the location of the component on the AUT's GUI. However, this technique requires testing scripts to be re-recorded if the GUI changes since scripts are regularly coupled to screen coordinates, and the effectiveness of the script relies on the ability to represent complex gestures. Besides, the record and replay technique requires a significant effort for the collection of adequate numbers and various user interactions to acquire effective test suites (Singh et al., 2014).

SPAG-C (Lin et al., 2014) and SPAG (Lin et al., 2013) implement a record and replay approach, which depends on the image comparison of screen-shots to generate accurate and reusable tests oracles. However, the replay process needs manual operations.

RERAN (Gomez et al., 2013) primarily focuses on the record and replay task, records low-level system events by leveraging the Android get events utility, and generates a replay script for the same device. However, generated scripts are not appropriate for replay on multiple devices due to the dependence of recorded interactions on-screen coordinates.

ACRT (Liu et al., 2014) extends Robotium to execute the recorded test scripts. ACRT generates tests that are dependent on sleep commands, making the tests slow and inaccurate. Besides, there is limited support for interactions and oracles, and the tool does not take into cognizance how GUI elements can be identified.

TestDroid (Kaasila et al., 2012) records user actions by tracking the UI components and automatically generates test scripts with calls to Robotium API. However, TestDroid is limited when the AUT depends on speech, movement, or gesture input.

Barista (Fazzini et al., 2017) generates tests in a visual and intuitive way. It records user interactions with any Android application based on screen coordinates, element ID, and text, and then automatically generates oracles. Those interactions and oracles convert into Espresso typescript for later execution. Barista requires the tester to manually write down test code, creating sequences of interactions with the components of the GUI.

2.4.2 Random based

The state of practice in automated Android app GUI testing is random-based. Random based approach explores AUT by generating random actions. There is a strong probability that the actions already selected will be selected again, which could eventually lead to lower code coverage. Random testing could generate test cases that are redundant, inefficient, and very difficult to comprehend and manage (e.g., for debugging purposes). Recent empirical studies of current GUI testing tools by (Choudhary et al., 2015; Wang et al., 2018) claim Android Monkey, a random testing tool for Android is the best among the existing test generation tools. However, it generates a large number of inputs efficiently, which can flood the AUT's GUI. Also, it does not generate inputs that need human intelligence (e.g., constructing valid passwords, playing and winning a game), and it does not generate highly specific inputs that control the application's functionality. Besides, it does not keep track of part of the application that has already been covered and is likely to generate redundant events. To overcome these limitations, Dynodroid (Machiry et al., 2013) uses three different heuristic exploration strategies, including two different random techniques and an Active Learning technique aiming at the execution of all the different events. Dynodroid selects relevant events to the application's current states and repeats the process in the observe-select-execute cycle. The advantages of Dynodroid are allowing both automated and manual input generation. However, Dynodroid uses instrumentation to infer relevant events to guide exploration.

PUMA (Hao et al., 2014) includes a generic UI Automator that uses a similar approach as an Android monkey but differs in its design, which uses dynamic analysis to trigger changes in the environment during app execution. It redefines the state pattern to generate events, but they are highly incompatible with the recent Android framework.

SmartMonkey (Haoyin, 2017) 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 (Chen et al., 2010). 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.

AutoDroid (Adamo, Nurmuradov, et al., 2018) uses a combinatorial based testing approach through a greedy algorithm for selection and execution events.

Hu et al. (Hu & Neamtiu, 2011) developed an approach based on Android monkey for automatically detecting UI crashes. It instruments AUT's source code and then automatically generates and performs test cases that are shown in a collection of log files for subsequent analysis. It does not use a structured approach to write test cases and relies on the pseudo-random user events created by Android Monkey.

Amalfitano et al. (Amalfitano et al., 2011) describe a GUI crawling-based approach that leverages completely random inputs to generate unique test cases.

AppDoctor (Hu et al., 2014) uses an approximate execution approach, which performs random testing by generating low-level event handlers in the code to simulate high-level user events. AppDoctor focuses on specific bugs that may cause crashes. It targets applications that use standard widgets and support standard actions. AppDoctor speeds up testing and automatically classifies most reports into bugs or false positives.

2.4.3 Model-based

Model-based testing is the most populous approach used for automating GUI testing. The GUI of the AUT is modeled, and appropriate tests are generated from the model. The generated test cases are used to validate if the AUT met the functional requirements (Su, 2016). A model-based exploration can be guided to specific unexplored parts using a systemic strategy such as depth-first exploration, breadth-first exploration, or hybrid (Azim & Neamtiu, 2013), or a stochastic model (Su et al., 2017). The model-based technique has encountered difficulties in inaccurate modeling. More specifically, dynamic behaviors in GUIs can generate inaccurate model or state explosion issues due to non-deterministic changes in GUIs. Hence, the model-based approach ignores the changes, finds the event unimportant, and then proceeds with the discovery differently. Explicitly, a GUI model that includes only a limited range of possible behavioral spaces can decrease the effectiveness of tests.

Stoat (Su et al., 2017) performs 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. Stoat randomly includes system events in the state model. The disadvantage of Stoat is that both steps need considerable time to execute.

FSMdroid (Su, 2016) constructs an initial stochastic model automatically for the AUT by using the static and dynamic analysis to identify UI events. GUICC (Baek & Bae, 2016) conducted a study of multi-level state representations to show that different levels of abstraction have an impact on the effectiveness of a modeling-based tool.

Amoga (Salihu et al., 2019) generates GUI models by using static and dynamic analysis for Android applications. It explores application behavior by implementing a crawling technique that uses the event list of the UI elements related to each event to exercise the event order at runtime dynamically. Amoga uses an augmented Dijkstra algorithm to reduce model crawling time.

APE (Gu et al., 2019) is built on top of Android monkey. It dynamically optimizes the statically defined state model to increase the effectiveness of Android Monkey. APE incorporates the application's behaviors by using a decision tree-based abstraction model that in each interaction depends on the feedback obtained at runtime during testing. An abstraction model can effectively balance the size and precision of the model.

TrimDroid (Mirzaei et al., 2016) uses the combinatorial testing approach. TrimDroid has four components: First, Model Extraction, which generates an Interface model that represents the app's GUI inputs, and Activity model that represents the app's GUI activities with its event handler. Second, Dependency Extraction, which identifies dependency between the GUI component and event handler. Third, Sequence Generation utilizes the Alloy Analyzer to generate event sequences that preserve the paths in the activity model. Fourth, Test-Case Generation which generates test cases.

PATS (Wen et al., 2015) conducts its test in a fine-grained framework on a set of parallel test nodes. The testing nodes use the black-box approach to create a section of the testing sequence. The slave nodes analyze the designated UI interface and dynamically decide on short-term test event sequences. The coordinator gathers these short-term event sequences and sends them to the slave nodes for further testing and generation of new short-term event sequences.

MonkeyLab (Linares-Vásquez et al., 2015) extracts events from the source code, traverses them, and generates GUI-based event scenarios. The event logs representing scenarios that are recorded and executed by the testers. These logs are mined to obtain event sequences described at the GUI level instead of low-level events; language models are derived using the vocabulary of feasible events. MonkeyLab uses two approaches, namely Back-off (BO) and interpolation (INTERP), for computing probabilities. The derived models are used to generate event sequences; the sequences are validated on the target device where infeasible events are removed for generating actionable scenarios.

MobiGUITAR (Amalfitano et al., 2014) is an enhanced version of AndroidRipper (Amalfitano, Fasolino, Tramontana, De Carmine, & Imparato, 2012) that dynamically reverse engineers the state-machine model from the executing application. MobiGUITAR implements a breath-based algorithm to pass through an application to create a task list composed of event sequences. The tasks generate UI events in a state model that can be used to generate a test case. MobiGUITAR can use either random or predefined constant input values during the exploration. However, it uses simple breadth-first exploration that restarts the app from the initial state to backtrack to previous states, which is time-consuming for most or real-world android apps.

A3E-Targeted (Azim & Neamtiu, 2013) 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.

ORBIT (Yang et al., 2013) includes an action detector module and a dynamic crawler. It uses a static analytic approach for inferring the actions from the source code. Orbit follows three main steps: (1) Identify the place where an action is instantiated or registered; (2) Locate the component on which the action is fired; (3) Extract the component identifier that can be used later by the crawler to identify the corresponding entity and fire the action. The crawling stage uses the dynamic GUI crawling built on Robotium (Reda & Josefson, 2014). Orbit has a well-defined crawling algorithm for the crawling stage, which is much better than the depth-first search on the UI states.

A2T2 (Amalfitano et al., 2011) builds a model of the AUT's GUI, based on a crawling-based approach. A2T2 contains three phases, which are the instrumentation phase used to instrument the Java code and detect the Java crashes during runtime. The GUI crawler phase extracts information regarding the GUI components' activity, the event handlers trigger events and intercept the application's crashes, then creates a GUI tree based on all this information. The test case generator phase generates test cases from the GUI tree for crash and regression testing, and they can be executed in the Android emulator.

TEMA (Takala et al., 2011) model the AUT's GUI by state machines model that generated manually, each model abstracts an individual view of the GUI with two separate state machines: an action machine which presents the high-level functionality with action words and state verifications; and a refinement machine implements action words and state verifications using keywords. Then, TEMA generates and executes test cases automatically.

2.4.4 Active Learning

This technique is used to tackle the shortfalls of the model-based testing technique. In this technique, an active learning algorithm is employed with a testing engine to learn the GUI application model. The active learning algorithm will guide the generation of user input sequences concerning the model (Amalfitano et al., 2015a).). However, this technique does not reach good coverage levels (Amalfitano et al., 2015b). Active learning may exploit the GUI's systematic exploration strategies, such as those emulating well-known graph exploration algorithms such as Breadth-first search (BFS), Depth-first search (DFS), or both. The following tools adopted the model learning technique.

SwiftHand (Choi et al., 2013) adopted dynamic analysis and machine learning to infer an application model. The model then uses the inputs to execute the application and explore unexplored states without restarting the application to reach all the screens. Thereby minimizes the restart overhead. SwiftHand learns an approximate GUI model from the execution traces generated during the testing process. It then uses the learned model to select user inputs, which applies to previously unexplored states. Then the learned model is expanded and refined when it identifies discrepancies between the previously learned models.

A3E-Depth-First (Azim & Neamtiu, 2013) allows Android applications to be analyzed progressively while running on the actual devices and without needing access to the source code. A “Depth-First Exploration” strategy is applied in a way that imitates a user’s actions to explore activities. The main challenge of using A3E is its dynamic approach of representing activities since it represents every activity as an individual state without considering that the activity can exist in different states. This leads to missing some behaviors of the app since not all states of the activities are not explored.

DroidCrawle (Wang et al., 2014) automatically traverses the GUIs with a depth-first traversal algorithm. DroidCrawle communicates with the Android device to realize the raw information of the current GUI for recognizing the GUI components of the application. DroidCrawle sends user events to the device to cause GUI transitions automatically, which can reduce human labor.

Cadage (Zhu et al., 2015) dynamically constructs a GUI model with the GUI state for the AUT. Cadage uses a breadth-first algorithm to analyze unexecuted GUI event handlers and accepts a probabilistic algorithm to select a GUI input as a test event. Cadage has four parts. (1) The Inference obtains enabled events from the current screen according to the properties of the GUI widgets. (2) The Selector chooses the test event from the fire event. (3) The Executor sends the test cases selected by the Selector. (4) The modeler retains its GUI model by retrieving the current status of the GUI or by making a new state.

CrashScope (Moran et al., 2016) 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.

Droidbot (Li et al., 2017) is an open-source testing tool that utilizes a depth-first exploration strategy to generate user events and system events under a black-box approach. It generates and executes test cases based on the state transition model constructed on the fly. 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.

2.4.5 Systematic based

The systematic technique focuses on the problem execution of generating complex sequences of events through concolic testing. Concolic requires constructing a symbolic model for the program execution environment. It is a white box and requires heavily instrumenting the application in addition to the framework. It is used to check the properties of the Android GUI application, and concrete execution identifies the conditions (logic) of a real application, and this avoids reaching unreachable states during the usage of the Android application (Sen, 2007). Symbolic execution automatically partitions the input domain such that each partition corresponds to the unique scenario of the program (e.g., execution of a unique program path). It, therefore, prevents redundant inputs from being generated and can generate extremely unique inputs. Concolic testing is applied to Android GUI testing in Collider (Jensen et al., 2013), ACTEve (Anand et al., 2012), and SIG-Droid (Mirzaei et al., 2015). However, some key factors are responsible for limiting the technique of the tools. First, the modeling of a real-world execution environment for Android applications is difficult because it contains sensors, networks, and cloud services. Secondly, Collider (Jensen et al., 2013) and ACTEve (Anand et al., 2012) are not scalable as in black-box testing to real-world applications because of the notorious path explosion problem (Ravindranath et al., 2014). Thirdly, ACTEve explores the application from its entry point, without targeting specific parts of the application code, in contrast to Collider, which detects event sequences that reach a given target line in the application code. Finally, Symbolic execution techniques are instances of white box testing which often require access to the application source code and thus not applicable for many proprietary applications.

ACTEve (Anand et al., 2012) generates a sequence of events from single events using concolic testing. The idea is to track events from the beginning to the point where they are finally handled. ACTEve has four components: Instrumenter, Runner, Concolic testing engine, Subsumption analyzer. The advantages of ACTEve are resolving the Path-explosion problem of the ALLSEQs algorithm. However, that may not be sufficient to handle apps that have significantly more paths such as if the app has many widgets (e.g., a virtual keyboard). Besides, ACTEve is easily portable to different versions of Android but still not cross-platform compatible.

Collider (Jensen et al., 2013) generates event sequences that can reach a given target line in the application, which was not reachable with other automated testing techniques like a simple crawler and the Android Monkey. Collider includes two phases. First, the target agnostic symbolic summarization phase provides the summary for each event handler by performing the Concolic execution to infer the path conditions and symbolic states. Secondly, the sequence generation phase with the event handler, together with the application's UI model, builds an event sequence that leads from the entry state of the application to the target. Each path is extended incrementally by searching for an event handler that can be triggered in front of the path to satisfy some of the path constraints.

SIG-Droid (Mirzaei et al., 2015) was built based on Java Pathfinder and used the byte-code interpretation of AUT. SIG-Droid relied on symbolic execution and combining inputs with an automatically extracted GUI model from the source code. SIG-Droid includes three components. First, Model Generator analyzes the source code of AUT and builds two models; the Interface model presents the app’s GUI input and the widgets, and the behavior model captures App’s event-driven behavior and the relationships among the event generators with handlers. The second component is the Symbolic Execution Engine, and the third component is the Test Case Generator.

2.4.6 Search based

This approach has been applied in EvoDroid (Mahmood et al., 2014) and Sapienz (K. Mao et al., 2016). The limitation of this approach has a high computational cost. Search-based tools are computationally expensive and do not scale in practice (Choudhary et al., 2015).

EvoDroid (Mahmood et al., 2014) is the first testing tool that uses an evolutionary algorithm. It combines model-based with search-based techniques for generating high coverage GUI test cases from the extracted model. The extracted model is based on a static analysis of the manifest and XML configuration files, and a call graph model is based on a code analysis using MoDisco (Eclipse, 2010). It uses these models to guide the process of computational search.

Sapienz (K. Mao et al., 2016) 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 uses the specific GUIs and complex sequences of input events with a pre-defined pattern. This pre-defined 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.

2.4.7 Reuse based

Reused based technique does not generate test cases and relies on injecting existing test cases with event sequences that do not affect the outcome of the original test cases.

Thor (Adamsen et al., 2015) executes test cases from Robotium (Reda & Josefson, 2014) or Espresso (Google, 2019e) on Android apps in adverse conditions. However, Thor does not generate test cases and relies on injecting existing test cases with event sequences, which do not affect the outcome of the original test cases. Thor event sequences include (1) Activity state changes (Pause-Resume, Pause-Stop-Restart, and Pause-Stop-Destroy-Create), (2) manipulation of the audio manager.

ExtendedRipper (Amalfitano, Fasolino, Tramontana, & Amatucci, 2013) is an exploration-based technique that uses a dynamic analysis in Android apps. In this technique, the event pattern includes many 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.

2.5 Research Gaps and Limitations

This section presents open research issues and challenges in the Android GUI testing domain that include reproducible test cases, test oracle, test input generation, test coverage, crashes diagnosis, and fragmentation.

2.5.1 Reproducible Test Cases

The ability to create a reproducible test is essential for test case generation tools because a developer needs to reproduce the test cases. When a crash is detected, the developers usually need to reproduce the crash at least twice before the fault 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 like Android Monkey produce excessively long test sequences that are infeasible to be reproduced by a human, which makes them the least favorable option (Arcuri, 2011). 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.

2.5.2 Test Oracle

Test automation increases the overall effectiveness of the testing process by increasing the coverage and reducing the testing time and cost. Extensive research focuses on developing tools and techniques in order to support automated GUI testing of Android apps. However, these studies mainly target the generation of test cases and do not include test oracle to the automation process, thus leaving the testers to add test oracles to those test cases manually. This is considered an intensive and insufficient process. All of that can compromise the efficiency of test cases (Barr et al., 2014; Zaeem et al., 2014).

2.5.3 Test Input Generation

Test case generation tools produce relevant inputs to exercise apps behavior (Memon, 2002). Android apps can sense and respond to numerous inputs from system interactions (Rubinov & Baresi, 2018; Yu & Takada, 2016). Interaction with system events includes receiving SMS notifications, app notifications, or events coming from sensors. These experiences are events that must be addressed in testing Android apps, which effectively increases the complexity of testing an app. Most of the recent testing tools for Android apps focus on UI events. Thus, they make it difficult to identify other defects in the changes that can be preferred by the context in which an app runs.

Moreover, tools should allow inputs to be manually provided, such as text data input. Specific inputs like logins and passwords can only explore certain behaviors. However, these behaviors may be challenging to reproduce randomly or using systematic techniques. Tools like Dynodroid and GUIRipper allow users to manually input values the tool can use during its analysis. This research argues that human knowledge for manual input should be integrated into test automation for a more effective test.

2.5.4 Test Coverage

Test coverage is another challenging factor for researchers during the automation of Android app testing. Researchers cannot effectively attain high code coverage to maximize test efficiency. For Android applications, there are a vast number of potential combinations of functions and transitions between them. This makes testing all potential combinations time-consuming and ineffective for large systems. Moreover, existing GUI testing tools cannot effectively explore too many app functionalities. The reason is that some functionality can only be reached through a particular sequence of events. For instance, random based are likely to detect more faults because of their unexpected nature, while systematic based tools may not expose the same number of faults unless they coverage 100%. Model-based tools have worse coverage performance than random approaches due to their difficulties in constructing a precise model and state explosion issues (Choudhary et al., 2015). For example, a finite state model that includes system events may not even exist for real-world apps (Baek & Bae, 2016). Code coverage is useful means of effectiveness evaluation for automated GUI testing tools. However, it is practically impossible to give an absolute estimation of tools' effectiveness in terms of code coverage since all testing tools are evaluated on a different set of Android applications and testing environment. Thus, there is a need to empirically analyze and compare existing GUI testing tools in terms of code coverage.

2.5.5 Crashes Diagnose

Test case generation tools are unable to provide a comprehensive, comprehensible crashes 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 crashes. 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. Both Crashscope (Moran et al., 2016) and DroidWalker (Hu et al., 2017) were the tools that can generate reproducible test scenarios. 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 (Su et al., 2020).

2.5.6 Fragmentation

One of the significant problems faced by Android developers is fragmentation. The term fragmentation has been used to describe variability due to the diversity of mobile device vendors. Test generation tools for Android should support a variety of devices with different hardware characteristics and use various releases of the Android framework (API versions) so that developers could assure the proper functioning of their applications on nontrivial sets of configurations. Thus, Droidbot and Android Monkey can be run on different versions of the Android framework.

One can represent configuration sets as a testing matrix that combines 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 (Kowalczyk et al., 2018). More studies are required to study the non-deterministic app behaviors.

CHAPTER 3: EXPERIMENTAL ANALYSIS ON TEST CASE GENERATION TECHNOLOGIES

This chapter investigates the effectiveness of the test 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 generation tools to detect a crash. It is critical to highlight its effectiveness due to its significant effects on testing time, effort, and computational cost. There are a number of test generation tools for Android apps available in the literature. These tools share the common goal of exploring apps' behavior to discover potential faults by generating user and system events. Thus, this chapter evaluates the effectiveness of six test input generation tools for Android apps on 50 apps downloaded from the repositories of F-Droid and AppBrain. The tools were evaluated and compared based on the activity coverage, method coverage, and capability in detecting crashes. The empirical case study method was adopted.

The remainder of this chapter is divided as follows: Section 3.1 highlights the followed case study design, while section 3.2 enumerates the execution steps outlined in the case study. Section 3.3 analyzes and discusses the findings. Section 3.4 presents research problems found. The possible threats to the validity of the results are discussed in Section 3.5, before concluding the findings in section 3.6.

3.1 Case Study Design

This analysis employs the empirical case study method that is used in software engineering, as reported in (Kitchenham et al., 2002; Perry et al., 2004). The method involves three major steps: (1) specify case study objectives, (2) select a case study that has data collection, and (3) a case study design for execution and evaluation.

3.1.1 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 case study 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?

3.1.2 Case Study Criteria

Coverage criterion is one of the critical testing requirements that some elements of the app should be covered (Morrison et al., 2012). 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 (Azim & Neamtiu, 2013). 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 (Azim & Neamtiu, 2013; Koroglu et al., 2018). 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 (Dashevskyi et al., 2018). 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 (Azim & Neamtiu, 2013; Choudhary et al., 2015; Dashevskyi et al., 2018; Koroglu et al., 2018).

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 a greater number of screens have been explored, and thus it will be more exhaustive for the app exploration (Azim & Neamtiu, 2013; Hu et al., 2014; Koroglu et al., 2018).

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.

3.1.3 Apps Selection

For the experimental analysis, 50 Android apps were chosen from F-Droid (F-Droid, 2010) and AppBrain (AppBrain, 2009). Table 3.1 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). 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) App permissions required: In this study, only apps that require at least two of the 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.

3.2 Case Study Execution

The selected tools for the experiment are Sapienz (K. Mao et al., 2016), Stoat (Su et al., 2017), Droidbot (Li et al., 2017), Humanoid (Li et al., 2019), Dynodroid (Machiry et al., 2013), and Android Monkey (Google, 2019j). 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, A3E, PUMA, and ACTEve in previous evaluation (Choudhary et al., 2015), (Wang et al., 2018). Table 3.2 provides an overview of the existing test input generation tools that support system events generation in the literature. The table reports all these tools and classifies them according to their features. All the tools were installed on a dedicated machine before starting the experiments.

Table 3.1: Overview of Android applications selected for testing

No	App name	Package name	Version	Category	Activity	Method	LOC
1	Aard	aarddict.android	1.5	Books	6	438	65511
2	Open Document	at.tomtasche.reader	1.6	Books	3	446	13796
3	Bubble	com.nkanaev.comics	4.1	Books	2	463	58397
4	Book Catalogue	com.eleybourn.bookc atalogue	2.1	Books	21	1548	11264
5	Klaxon	org.nerdcircus.androi d.klaxon	1.6	Communication	6	162	5733
6	Sanity	cri.sanity	2	Communication	28	1398	44215
7	WLAN Scanner	org.bitbatzen.wlansca nner	4	Communication	1	141	5441
8	Contact Owner	com.appengine.paran oid_android.lost	1.5	Communication	2	79	2502
9	Divide	com.khurana.apps.div ideandconquer	2.1	Education	2	195	25284
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.curr ency	4	Finance	5	148	5202
14	Mileage	com.evancharlton.mil eage	1.6	Finance	50	2486	92548
15	TimeSheet	com.tastycactus.times heet	2.1	Finance	6	198	7126
16	Boogdroid	me.johnmh.boogdroi d	4	Game	3	398	3726
17	Hot Death	com.smorgasbork.hot death	2.1	Game	3	365	28104
18	Resdicegame	com.ridgelineapps.res dicegame	1.5	Game	4	144	2506
19	DroidWeight	de.delusions.measure	2.1	Health & Fitness	8	411	13215
20	OSM Tracker	me.guillaumin.androi d.osmtracker	1.6	Health & Fitness	8	346	49335
21	Pedometer	name.bagi.levente.pe dometer	1.6	Health & Fitness	2	244	6695
22	Pushup Buddy	org.example.pushupb uddy	1.6	Health & 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	23294
25	Car cast	com.jadn.cc	1.5	Music & Audio	12	459	18127

Table 3.1: Continued

No	App name	Package name	Version	Category	Activity	Method	LOC
26	Ethersynth	net.sf.ethersynth	2.1	Music & Audio	8	168	1208
27	Jamendo	com.teleca.jamendo	1.6	Music & Audio	13	1046	30444
28	Adsdroid	hu.vsza.adsdroid	2.3	Productivity	2	1215	5080
29	Maniana	com.zapta.apps.maniana	2.2	Productivity	4	891	28526
30	Tomdroid	org.tomdroid	1.6	Productivity	8	840	29147
31	Talalarmo	trikita.talalarmo	4	Productivity	3	387	1350
32	Unit	info.staticfree.android.units	1.6	Productivity	3	547	22993
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	24601
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	16625
43	Terminal Emulator	jackpal.androidterm	1.6	Tools	8	994	24930
44	Timeriffic	com.alfray.timeriffic	1.5	Tools	7	709	28956
45	Addi	com.addi	1.1	Tools	4	2133	133448
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	19837

The experiment conducts on a 64-bit Ubuntu 16.04 physical machine with eight-cores (3.50 Gigahertz Intel Xeon ® CPU) and eight Gigabytes of RAM and uses an Android emulator x86 ABI image (KVM powered). 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 used in the experiment for 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 (Machiry et al., 2013) 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 minutes on each specified app.

To compensate for the possible impact of 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.

Table 3.2: Overview of Android test generation tools

No	Tool	Approach	Exploration Strategy	Events	Crash Report	Replay Scripts	Emulator/ Device	Available
1	**Humanoid** (Li et al., 2019)	Black-box	Deep Q Network	UI, System,	No	Both	Both	Yes
2	**AndroFrame** (Koroglu et al., 2018)	Black-box	Q-Learning-Based	UI, System, Context	-	Yes	Both	No
3	**DroidBot** (Li et al., 2017)	Black-box	Model-based	UI, System	No	Yes	Both	Yes
4	**SmartMonkey** (Haoyin, 2017)	Black-box	Random-based	UI, System, Context	-	-	-	No
5	**Stoat** (Su et al., 2017)	Black-box	Model-based	UI, Text, System, Context	Text	Yes	Both	Yes
6	**Sapienz** (K. Mao et al., 2016)	Grey-box	Search-based/ Random	UI, Text, System, Context	Text, Video	Yes	Emulator	Yes
7	**Crashscope** (Moran et al., 2016)	Grey-box	Systematic	UI, Text, System, Context	Text, Image	Yes	Both	No
8	**Dynodroid** (Machiry et al., 2013)	Black-box	Guided/Random	UI, Text, System, Context	Text, Image	Yes	Emulator	Yes
9	**ExtendedRipper** (Amalfitano, Fasolino, Tramontana, & Amatucci, 2013)	Black-box	Model-based	UI, Text, System, Context	No	No	Emulator	No
10	**A3E-Targeted** (Azim & Neamtiu, 2013)	Grey-box	Systematic	UI, System, Context	-	No	Both	No
11	**Android Monkey** (Google, 2019j)	Black-box	Random-based	UI, System	No	No	Both	Yes

3.3 Results and Discussion

This section examines the results based on the case study's objectives outlined in section 3.1.1. Case study questions were answered by measuring and comparing the number of crashes detected, the method coverage, and activity coverage achieved by each testing tool on selected apps in the experiments. Table 3.3 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 3.3: Statistics of results on apps by test generation tools understudy

Apps Under Test	**Method coverage (%)**						**Activity coverage (%)**					
	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
Aard	17.4	16.4	16.4	16.2	17.4	17.4	33	33	33	33	33	33
Addi	10.1	9.5	9.6	9.6	4.6	4.7	50	50	50	50	50	25
Adsdroid	56.0	56.0	56.0	56.3	56.1	56.0	100	100	100	100	100	100
Alarm Clock	24.6	64.5	43.9	62.7	24.9	17.9	60	60	60	60	20	40
Alogcat	72.9	75.9	52.8	72.9	67.3	46.2	100	100	100	100	100	100
Android Token	54.5	57.6	54.5	49.5	51.4	50.0	67	67	50	67	50	50
applicationsinfo	64.5	64.3	64.4	64.7	44.3	44.3	100	100	100	100	67	67
Battery Circle	81.5	81.5	81.5	84.0	79.0	79.0	100	100	100	100	100	100
Blockinger game	77.5	81.7	81.5	80.1	16.6	11.2	100	100	100	100	67	50
Boogdroid	13.0	10.1	13.0	16.7	15.6	15.3	100	33	67	67	56	33
Book Catalogue	31.7	4.0	33.0	32.9	43.4	32.7	41	5	48	38	43	29
Bubble	54.9	55.9	36.9	30.5	67.6	0.0	100	100	100	50	100	0
Car cast	44.9	46.2	41.6	43.2	34.0	29.8	75	67	67	67	47	72
Contact Owner	54.4	54.4	57.0	57.0	51.1	51.9	50	50	50	50	50	50
Currency	58.8	64.0	40.3	56.8	42.6	40.5	100	100	80	100	100	60
Dew Point	75.6	73.3	76.4	77.3	58.7	58.7	100	100	100	100	67	67
Divide	52.8	47.2	52.8	52.8	74.4	46.2	100	100	100	100	100	100
DroidWeight	74.1	62.9	69.8	70.6	72.7	72.7	50	38	67	75	58	38
drhoffmann	55.9	59.8	51.2	59.1	43.3	36.6	85	100	93	93	93	78
Ethersynth	66.7	65.1	50.0	57.7	66.7	64.3	100	100	100	100	88	63
FindMyphone	0.1	0.1	0.1	0.1	0.1	0.1	100	100	100	100	100	100
Hot Death	74.6	59.4	65.8	70.4	76.8	55.6	100	100	100	100	100	100
Jamendo	53.1	34.7	43.9	43.9	24.4	0.0	62	31	38	38	46	0
Klaxon	44.4	38.3	36.6	35.8	39.9	41.4	83	83	83	83	83	78
List my Apps	76.4	75.3	46.9	72.9	72.9	72.9	50	100	25	100	100	25
LolcatBuilder	32.9	27.8	32.9	32.9	25.3	25.3	100	100	100	100	100	100
Maniana	75.4	58.4	54.1	54.0	72.8	66.9	75	75	75	75	75	50
Mileage	30.7	16.0	25.6	25.5	25.0	27.6	44	25	38	40	22	27
Mirrored	33.3	39.7	47.0	47.0	32.4	30.1	75	75	75	75	50	50
MunchLife	66.7	66.7	66.7	66.7	59.0	59.0	100	100	100	100	100	100
Open Document Reader	54.4	42.2	45.3	45.3	70.6	36.8	33	33	33	33	33	33
OpenSudoku	62.5	54.5	40.3	40.3	42.2	38.1	50	30	30	30	50	50
OSM Tracker	44.2	62.9	48.1	48.3	50.6	43.1	75	71	83	88	92	50
Pedometer	70.1	69.7	61.5	61.5	83.2	70.4	100	100	50	100	100	50
Pushup Buddy	48.5	48.9	48.9	56.4	44.2	43.0	57	57	43	71	57	43
Raele.concurseiro	41.7	42.1	41.7	41.7	41.7	41.7	100	100	100	100	100	100
Resdicegame	62.5	48.6	62.5	53.5	47.9	47.9	100	100	100	100	100	100
Sanity	28.4	18.6	24.9	24.3	30.4	14.2	48	29	48	46	61	21
Sensor read out	30.2	30.0	30.0	30.0	27.8	30.0	100	100	67	67	67	67
Sensors2Pd	17.1	20.0	20.5	21.7	20.5	20.5	100	100	100	100	100	100
Talalarmo	88.1	88.1	82.2	88.9	90.4	71.9	100	100	100	100	100	100
Terminal Emulator	55.2	44.2	51.3	51.7	52.3	49.5	38	38	38	38	38	25
Timeriffic	63.8	50.5	57.1	57.1	59.8	54.9	86	52	57	57	71	29
TimeSheet	59.4	33.5	27.8	27.8	20.7	20.2	100	67	50	50	50	56
Tomdroid	36.1	51.9	37.5	37.0	40.8	0.0	63	63	75	75	63	0
Unit	69.9	55.6	52.5	52.5	69.1	54.8	33	33	33	33	67	33
Weather notifications	59.4	50.0	41.7	37.4	74.2	67.6	71	52	57	57	71	43
WLAN Scanner	66.0	63.1	65.2	65.2	61.7	61.7	100	100	100	100	100	100
World Clock	43.8	56.8	29.2	29.2	22.9	22.9	100	100	75	75	50	50
Overall average	**39.8**	**35.1**	**36.1**	**36.8**	**36.9**	**28.8**	**66.3**	**55.3**	**60.7**	**62.8**	**58.1**	**42.0**

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 (Saswat, 2015). Ella is a binary instrumentation tool for Android apps. From Table 3.3, it 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., (K. Mao et al., 2016) 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 bytecode at the compile time to track the execution of the code at runtime. Sapienz used EMMA (Roubtsov, 2005) white-box instrumentation tool to achieve full statement coverage, while Ella (Saswat, 2015) 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 like 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.

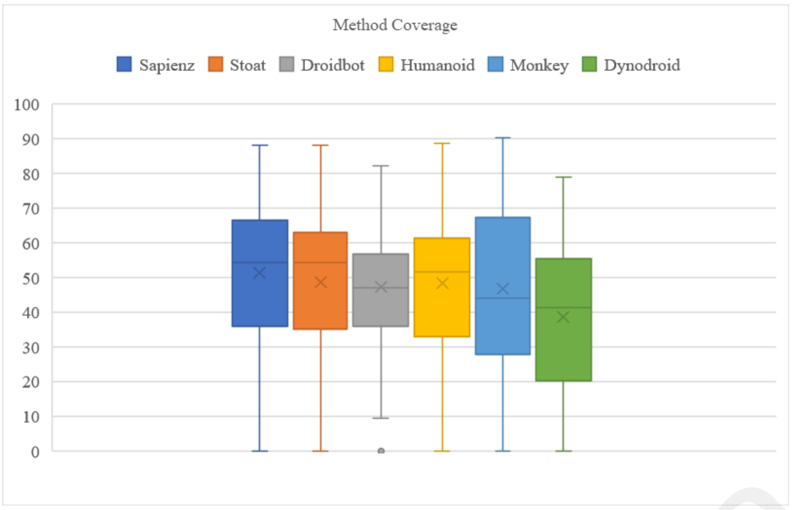


Figure 3.1: Variance of method coverage achieved across apps and three runs.

To further investigate experiment findings, Figure 3.1 shows 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. 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 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.

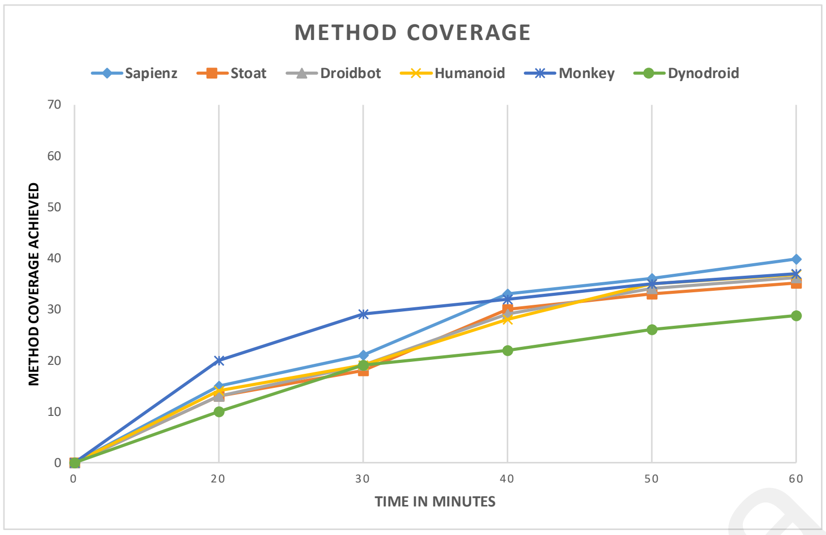


Figure 3.2: Progressive method coverage achieved across apps and three runs.

Figure 3.2 reports the progressive coverage of each tool over the time threshold of 60 minutes. The progressive average coverage of each of the test input generation tool was calculated across all 50 apps for every 20 minutes. The final coverage achieved was compared and reported. In the first 20 minutes, the coverage for all testing tools was observed to be increased rapidly as the apps were just started. At 40 minutes, 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 minutes of execution.

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, which was similar to the previous experiment on method coverage. Due to its ability to explore and optimize the test sequences as reported by (K. Mao et al., 2016), 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, 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 Android Monkey produced more inputs than other approaches, 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 AUT (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. 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 was able to explore more than 44% of activity coverage on the Mileage app.

Figure 3.3 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.

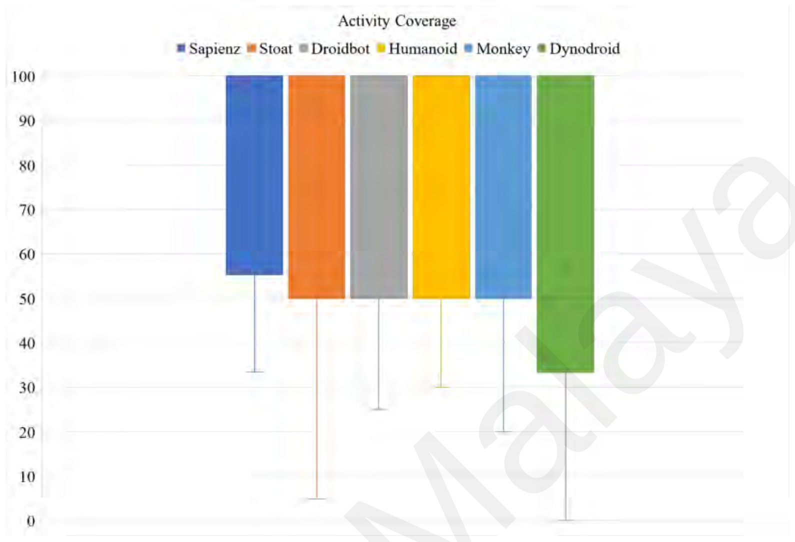


Figure 3.3: Variance of activity coverage achieved across apps and three runs.

From Figure 3.3, 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 out 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.

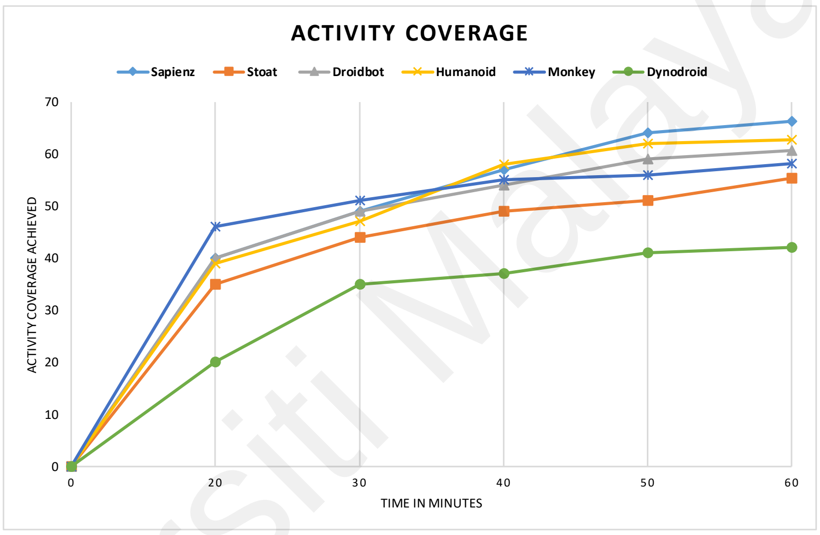


Figure 3.4: Progressive activity coverage achieved across apps and three runs.

As shown in Figure 3.4, 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 minutes, but the fastest convergence of each tool was different. From the results, Android Monkey, Humanoid, and Sapienz had the highest coverage at 20 minutes, 40 minutes, and 50 minutes, respectively. Other tools such as Stoat, Droidbot, and Dynodroid did not achieve the highest coverage before the final convergence time of 60 minutes. Stoat required more execution time because it has an initial phase to construct an app state-model for the generation of the test case. 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. As 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 defines 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 (Google, 2019c) 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's 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. (Su et al., 2017) 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 (Moran et al., 2016) 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 (Dashevskyi et al., 2018).

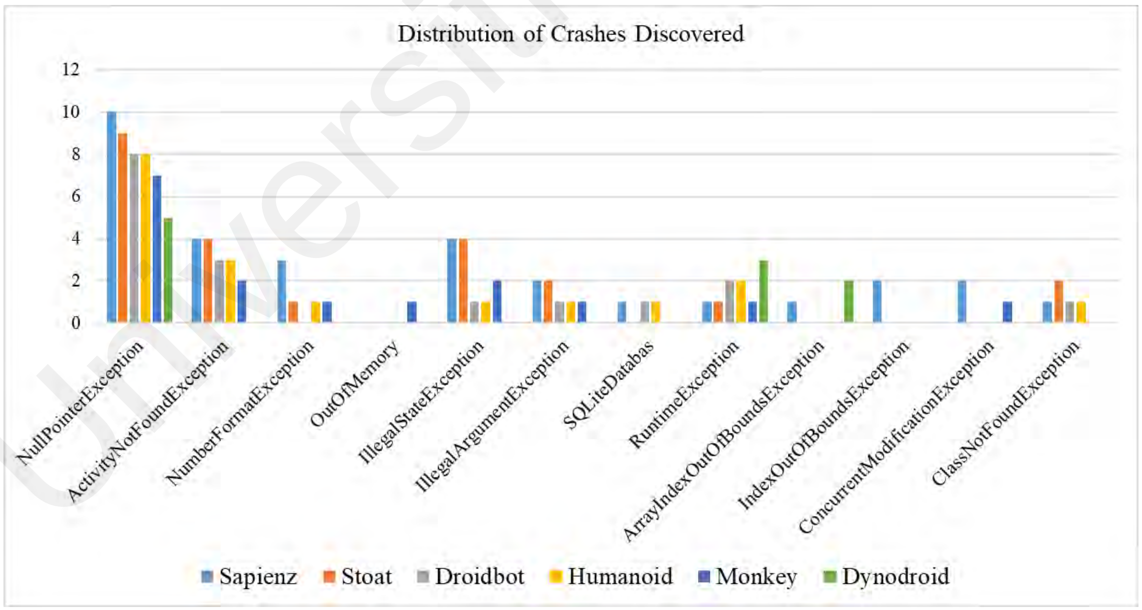


Figure 3.5: Distribution of Crashes Discovered.

Figure 3.5 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 (Harman et al., 2012). However, Sapienz used the Android Monkey input generation, which continuously generated events without waiting for the effect of the previous event. Sapienz triggered many Class-Cast-Exception and Concurrent-Modification-Exception. All of them were found through trackball and directional pad events. However, these crashes were insignificant. The reason is that trackballs and directional pads were unavailable on new Android phones. Besides, Sapienz triggered numerous SQLite Exceptions on the Jamendo app for all three runs. The exceptions majorly concern querying on multiple non-existent tables in the app's SQLite database. Because the apps depend majorly on the SQLite database and do not adequately deal with related exceptions, these destructive SQL queries are frequently triggered by the app's multiple locations. In effect, the fatal SQL queries cause multiple stack traces. During the testing, none of the other tools triggered SQLite exceptions, reflecting the several mistakes of using Android's default database. The only possible explanation is that initiating such crashes requires specific preconditions. A good example of such specific preconditions is forcibly terminating the app during initialization, which involves SQL operations for creating these tables, which the other tools might not create. Moreover, Android Monkey detected some stress testing bugs such as Illegal State Exceptions from the synchronizations between List Views and their data adapters, Illegal Argument Exceptions from the mismatches that result from service binding or nonbinding as a result of the rapid switches of activity lifecycle callbacks, and Out-Of-Memory-Errors (Su et al., 2017). Out-Of-Memory-Errors may occur when the app attempts to load a large-size located on 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 the SD card (Su et al., 2020). 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. As 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 app like "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 like ArrayIndexOutOfBoundsException and NullPointerException.

Table 3.4 shows the statistics of crash results on apps by test generation tools. Sapienz triggered on average 32 unique crashes in 26 apps, followed by Stoat detected 25 unique crashes in 19 apps. Droidbot and Humanoid triggered 19 and 20 unique crashes respectively, on the17 apps.

Table 3.4: Statistics of crash results on apps by test generation tools understudy

Apps Under Test	# of Unique Crashes					
	Sapienz	Stoat	Droidbot	Humanoid	Monkey	Dynodroid
A2DP Volume	1	0	2	2	0	0
Aard	1	0	1	1	0	0
Addi	1	1	1	1	1	1
Adsdroid	1	1	0	0	1	0
Alarm Clock	1	0	2	2	0	0
Alogcat	0	0	0	0	0	0
Android Token	0	0	0	0	0	0
applicationsinfo	0	0	0	0	0	0
Battery Circle	0	0	0	0	0	0
Blockinger game	0	0	0	0	0	0
Boogdroid	1	0	1	1	1	1
Book Catalogue	1	1	1	1	1	1
Bubble	2	2	0	0	1	0
Car cast	2	3	1	1	2	1
Contact Owner	1	1	1	1	1	1
Currency	1	1	0	0	0	0
Dew Point	2	1	1	1	0	1
Divide	0	0	0	0	0	0
DroidWeight	1	0	0	0	0	0
drhoffmann	2	2	1	2	2	1
Ethersynth	1	1	0	0	0	0
FindMyphone	0	0	0	0	0	0
Hot Death	0	1	0	0	0	0
Jamendo	2	1	1	1	0	0
Klaxon	0	0	0	0	0	0
List my Apps	0	0	0	0	0	0
LolcatBuilder	0	0	1	1	1	0
Maniana	0	0	1	1	0	0
Mileage	1	1	1	1	1	1
Mirrored	1	0	0	0	0	0
MunchLife	0	0	0	0	0	0
Open Document Reader	0	0	0	0	0	0
OpenSudoku	1	1	1	1	1	1
OSM Tracker	1	2	1	1	1	1
Pedometer	1	2	0	0	1	0
Pushup Buddy	0	0	0	0	0	0
Raele.concurseiro	0	0	0	0	0	0
Resdicegame	0	0	0	0	0	0
Sanity	1	1	0	0	0	0
Sensor read out	1	0	0	0	0	1
Sensors2Pd	2	1	1	1	1	0
Talalarmo	0	0	0	0	0	0
Terminal Emulator	1	1	0	0	0	0
Timeriffic	0	0	0	0	0	0
TimeSheet	0	0	0	0	0	0
Tomdroid	0	0	0	0	0	0
Unit	0	0	0	0	0	0
Weather notifications	1	0	0	0	1	0
WLAN Scanner	0	0	0	0	0	0
World Clock	0	0	0	0	0	0
Overall average	**32**	**25**	**19**	**20**	**17**	**11**

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 the 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 in an hour and explored the same activities repeatedly with no new coverage. An example of AUTs (A2DP Volume) is presented in Table 3.1. Android Monkey clicked the back button to return to the main activity and the cycle repeats. Such repeated actions caused redundant explorations and occupied 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 minutes as seen in Figure 3.2 and Figure 3.4. Hence, Sapienz explored visited states and generated more event sequences.

Table 3.5: Experimental results to answer research questions

Tools	Activity Coverage (%)	Method Coverage (%)	Number of crashes	Max Events Number
Sapienz	66.3	39.8	32	6000
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

Table 3.5 shows the maximum number of event sequences required by each tool to achieve the results. On average, Stoat, Droidbot, Humanoid, and Dynodroid generated a total of 3000, 1000, 1000, and 2000 events in an hour, respectively. Sapienz produced 6000 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.

The 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 (Xie & Memon, 2006). 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. (Baek & Bae, 2016) 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 event sequences increased the coverage and crash detection, however, more extended event sequences have many disadvantages such as high redundancy, high computational costs, and are difficult to interpret manually.

3.4 Research Problems Found

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. Lastly, some of the functions that should be considered by other tools were highlighted.

3.4.1 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. Also, 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.

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.

3.4.2 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 (K. Mao et al., 2016). 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. The 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 serious crashes because it needs to return the app to a new clean state before starting a new testing script.

3.4.3 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.

3.4.4 Access Control

One of the key aspects of software security is Access Control (Kayes et al., 2020). There are many mechanisms of Access Control in the literature. These Access Control mechanisms exist to restrict access to a software system's security-sensitive resources and functionalities (Borges & Zeller, 2019; Sadeghi et al., 2017; Shebaro et al., 2014). Mobile device resources can collect sensitive data, and they may expose the user to security and privacy risks if apps misuse them without the user's prior 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 employs a permission-based security mechanism to tackle access to sensitive data and potentially dangerous device functionalities. However, the process is not always a straightforward task to properly use this permission-based security mechanism. The behavior of the Android app may change depending on the granted permissions. It needs to be tested under a wide range of permission combinations (Sadeghi et al., 2017). Test 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.

3.4.5 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. Like 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, test generation tools for Android apps in the literature are typically impractical for developers to use due to the instrumentation and the platform required.

3.5 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 and AppBrain repositories, which are commonly used in Android GUI testing studies. Section 3.1.3 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.

3.6 Conclusion

This chapter presents an empirical analysis 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 chapter, 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.

CHAPTER 4: PROPOSED SOLUTION

This chapter presents an overview of the proposed approach for generating GUI test cases for Android Apps. The proposed solution is derived from the review between reinforcement learning, and test case generation approaches explained in chapter 2 to take advantage of both the randomly based approach and model-based approach. This approach generates user and system inputs that discover unexplored states of the app and uses the execution of the app on the generated inputs to construct a state-transition model. Instead of randomly selecting the actions, the test generator learns how to act in an optimal way that explores new states by using new actions to gain more rewards to maximize instruction coverage, method coverage, and activity coverage with minimizing redundant execution of events sequence.

The remains of this chapter are organized as follows: Section 4.1 presents the background of reinforcement learning, its technologies, and techniques. Section 4.2 introduces the adaptation of reinforcement learning techniques in GUI testing for Android apps. Section 4.3 justifies the reason for adopting the Q-Learning technique. Section 4.4 presents the implementation of the proposed approach while Section 4.5 highlights the significance of the proposed approach. Finally, Section 4.6 concludes this chapter.

4.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning. Unlike other branches like supervised and unsupervised learning, its algorithms are trained using reward and punishment to interact with the environment. It is based on the concept of behavioral psychology that works on interacting directly with an environment which plays a key component in Artificial Intelligence. As represented by the Markov Decision Process (MDPs), RL is a problem emerging in many real-world scenarios and provides mathematical frameworks that allow for modeling decision-making. It defines an environment's state, the action that the agent can take, the reward, and its expectation for the action and the next state after executing the action. However, the most important characteristic of an MDP is that the states' transition and reward function depend only on the current state and executed action. In an actual situation, the agent cannot precisely predict those functions (reward and state transition). If these functions are not known, the use of valued-based methods like Q-Learning is used to measure actions and predict them without knowing how good the actions are.

The major components of RL are the agent and the environment. The agent serves as an independent entity that performs unconstrained actions within an environment to achieve a specific goal. The agent performs an activity on the environment and uses trial-and-error interactions to gain information about the environment. There are four other basic concepts in the RL system along with agent and environment: (i) a policy, (ii) a reward, (iii) action, and (iv) state. The state describes the present situation of the environment and mimics the environment's behavior. For example, this gives rise to a current situation and action. The model might predict the resultant next state and the next reward. Models are used to plan and decide on a course of action by considering possible future situations before they are experienced. Similarly, the reward is an abstract concept to evaluate actions. Reward refers to immediate feedback after acting. The policy defines the agent approach to select an action from a given state. It is the core of the RL agent and sufficient to determine behavior. In general, policies may be stochastic. An action is a possible move in a particular state.

An analysis by McKinsey Global Institute Research (Chui et al., 2018) on the application of RL revealed that there are about 400 use cases across 19 industries and nine business functions highlight the significance and the broad use of advanced RL techniques ranging from advanced electronics/semiconductors, aerospace and defense, agriculture, automobile and assembly, banking, healthcare system, high tech, oil and gas, pharmaceuticals, public and social sectors, telecommunication, transport and logistics, travel and insurance companies.

The application of RL in the present world cannot be overemphasized. Mao et al., (2016) designed RL algorithms that allocate and schedule limited computer resources to different tasks that seem challenging and require human-generated heuristics. Arel et al. (2010) used the RL framework to obtain an efficient traffic signal control policy targeted to minimize average delay, congestion, and the likelihood of intersection cross-blocking. Bu et al., (2009) proposed RL for the autonomic configuration of a multi-tier web system; however, it adopted other policy initialization techniques to remedy the large state space. RL has been applied in optimizing chemical reactions that outperform a state-of-the-art algorithm, and this study optimized chemical reactions using Markov Decision Process (Zhou et al., 2017).

RL is a mainstream technique used to solve different games and sometimes achieve super-human performance. Researchers have adopted RL techniques to developed approaches for testing Android apps. Figure 4.1 presents the mechanism RL in the context of Android app testing. In the automated GUI testing, AUT is the environment; the state is the set of actions available on the AUT activity. The GUI actions are the set of actions available in the current state of the environment, and the testing tool is the agent. Initially, the testing tool does not know the AUT. As the tool generates and executes test event input based on trial-and-error interaction, the knowledge about AUT is updated to find a policy that facilitates systematic exploration to make efficient future action selection decisions. This exploration generates event sequences that can be used as test cases.

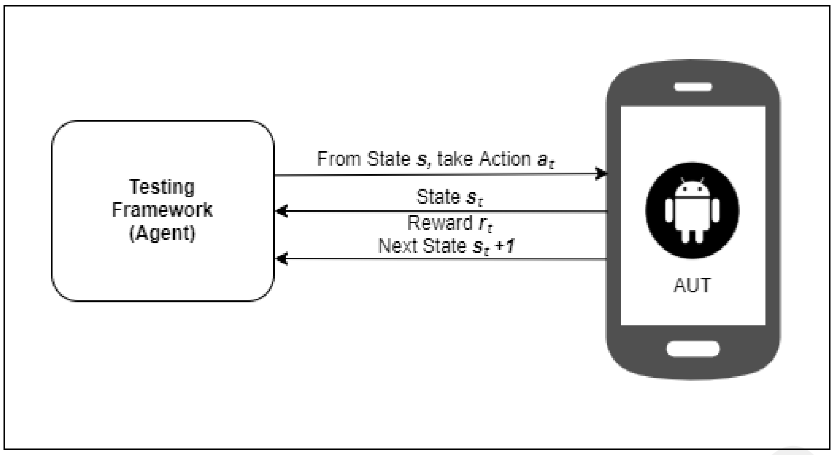


Figure 4.1: Reinforcement learning mechanism

There are two primary directions in solving RL problems: algorithms based on value functions/ Q-value $Q(s, a)$ and algorithms based on policy search (Arulkumaran et al., 2017). Q-Value-based algorithms update the value function based on an equation. Whereas the policy-based estimates the value function with a greedy policy obtained from the last policy improvement. Most RL technique follows a step-by-step pattern. At each time step t, it observes the environment's state S_t and takes action A_t based on its policy π. The environment then transitions to a new state S_{t+1} based on S_t and A_t, and it also outputs a scalar reward R_{t+1} as feedback that the agent then uses to update its knowledge. The agent learns a policy that maximizes the expected cumulative reward of a sequence of actions in the environment that are finally used as test cases.

Several exploration strategies have been proposed by integrating mathematical approaches, such as Epsilon-greedy ($\in -greedy$) policy and Upper confidence bound (UCB). The epsilon-greedy policy aims to identify a possible way and keep on exploiting it greedily. The agent randomly explores with probability $\in$ and selects the action with the highest Q-value in the current state with probability $1- \in$. The random action is useful for exploration, but it might also lead the agent to try out actions that will not give a good reward. Moreover, it explores too much because even when selected action seems to be the optimal one, the policy keeps allocating a fixed ratio of the time for exploration, thus missing opportunities and increasing total regret. UCB policy proposed by Auer et al (2002) for multi-armed bandit that achieves regret that grows only logarithmically with the number of actions taken. It enhances the exploration and minimizes the total regret. It explores more to reduce uncertainty systematically, but its exploration reduces over time. Thus, UCB attains greater reward on average than the Epsilon-greedy policy. There are several techniques of Reinforcement learning include Deep Q Network (DQN), Actor-critic, State-Action-Reward-State-Action (SARSA), and Q-Learning.

4.1.1 Deep Q Network

Deep Q Network (DQN) combined Q-Learning with a flexible deep neural network. It was tested on a varied and large set of deterministic Atari 2600 games, reaching human-level performance on many games. DQN uses the Neural Network to estimate the Q-value function. The network's input is the current state, while the output is the corresponding Q-value for each action. Although DQN has achieved huge success in higher-dimensional problems, such as the Atari game, the action space is still discrete. With many tasks of interest, especially physical control tasks, the action space can be continuous.

4.1.2 Actor-Critic

An Actor-Critic uses both the value function and the policy function, where the "Critic" estimates the value function. This could be the Action-value (The Q value) or State-value (the V value), and the "Actor" updates the policy distribution in the direction suggested by the Critic (such as with policy gradients). Actor-Critic's variants, namely the Asynchronous Advantage Actor-Critic (A3C) and the Advantage Actor-Critic (A2C). In essence, A3C implements parallel training where multiple workers in parallel environments independently update a global value function hence "asynchronous." One important benefit of having asynchronous actors is an effective and efficient exploration of the state space. A2C is synchronous as compared to A3C; it is a single worker variant of the A3C. A2C produces a comparable performance to A3C while being more efficient, although researchers are unsure if or how the asynchrony benefits learning.

4.1.3 State-Action-Reward-State-Action (SARSA)

State-Action-Reward-State-Action (SARSA), an on-policy Temporal Difference (TD) control method. A policy is a state-action pair tuple that maps the action to be taken at each state. This on-policy control method chooses each state's action during learning by following a certain policy (mostly the one it is evaluating itself, like in policy iteration). SARSA and Q-Learning are both policy control methods that evaluate the optimal Q-value for all action pairs. SARSA resembles Q-Learning to a lot of extents. The only difference between the two is that SARSA learns the Q-value based on the current policy's action compared to Q-Learning's use of the greedy policy. Equation (4.1) shows the update rule for SARSA:

$$\boldsymbol{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + a[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]} \tag{4.1}$$

4.1.4 Q-Learning

Q-Learning is also known as one of the most popular reinforcement learning techniques (Kaelbling et al., 1996). The "Q" in Q-Learning stands for quality representing how useful a given action is in gaining some future reward. It is an off-policy temporal difference control method. It is precisely like SARSA with the only difference that Q-Learning does not follow a policy to find the next action but instead chooses the action greedily. Similar to SARSA, it aims to evaluate the Q-values based on equation (4.2).

$$\boldsymbol{Q^*(s_t, a_t) = \max_{\pi} \sum_{t>0} (\gamma^t r_t | s = s_t, a = a_t, \pi)} \tag{4.2}$$

Q-Learning uses its Q-values to resolve RL problems. For each policy Π, the action-value function or quality function (Q-function) should be properly defined. Nonetheless, the value $Q\ \Pi\ (s_t;\ a_t)$ is the expected cumulative reward that can be achieved by executing a sequence of actions that starts with action a_t from s_t; and then follows the policy Π. The optimal Q-function Q^* is the maximum Q expected cumulative reward achievable for a given (state, action) pair over all possible policies. This is known as the Bellman equation. The Q-Learning algorithm uses equation (4.2) to estimate the value iteratively. Intuitively, if Q^* is known, the optimal strategy at each step s_t is to take action that maximizes the sum: $r\ +\ Q\ *\ (s_t + 1, a_t + 1)$, where r is the immediate reward of the current step, while t stands for the current time step, hence $t\ + 1$ denotes the next one. The discount value (γ) is introduced to control the long-term rewards' relevance with the immediate one.

Q-Learning is used to find an optimal action-selection policy for the given AUT using greedy policy and behaves using other policies such as Epsilon-greedy policy and UCB, where the policy sets out the rule that the agent must follow when choosing a particular action from a set of actions (Watkins & Dayan, 1992). There is an action execution that is immediately preceded to choose each action, which moves the agent from the current state to a new state. This agent is provided with a reward r upon executing the action a. The value of the reward is then measured using the reward function R. For the agent, the main aim of Q-Learning is to learn how to act in an optimal way that maximizes the cumulative reward. Thus, a reward is granted when an entire sequence of actions is carried out.

4.2 Automated GUI Testing with Reinforcement Learning

Researchers have developed approaches to automate test generations for Android apps. This section highlights the existing tools with corresponding approaches.

Mariani et al. (Mariani et al., 2012) proposed AutoBlackTest, the first Q-Learning-based GUI testing tool for Java desktop software. AutoBlackTest initially extracts an abstract representation of the current state of the GUI and generates a behavioral model. This model is updated according to the current state reached and the immediate utility of the action. Then the behavioral model is used to select the next action to be executed, and then to restart the loop. TESTAR (Esparcia-Alcázar et al., 2016), another Q-Learning-based tool, is used to generate GUI test sequences based on web applications. The Q-Learning algorithm provided significant performance with an adequate set of parameters.

GunPowder (Kim et al., 2018) is a test input generation tool for search-based test data generation using deep RL. GunPowder has been specifically developed for C applications and consists of three phases: (i) instrumentation, (ii) execution, and (iii) fitness evaluation. In the instrumentation phase, in the first step, it adds instrumentation codes that allow the tool to control and monitor the execution of the program. Subsequently, in the second step, the tool builds and executes the program, and in the third phase, the machine learning algorithm was applied to generate test inputs. Currently, the fitness function supported by GunPowder aims to improve branch coverage. Although not suitable for Android app testing, other studies that have adopted RL techniques for Android testing are shown in Table 4.1.

Table 4.1: Overview of Android test generation tools adopted RL

No	Tool	Action Selections	State Representation	Test Input generation	Coverage Criteria	Basis
1	**Humanoid** (Li et al., 2019)	Best 10 actions	Number of actions in activity	DQN	Line	Droidbot
2	**Androfram** (Koroglu et al., 2018)	Trained action	Activity ID & UI components	Q-learning	Activity	-
3	**(Adamo, Khan, et al., 2018)**	Action with highest Q-value	Number of actions in activity	Q-learning	Block	Appium
4	**(Vuong & Takada, 2018)**	Action with highest Q-value	Activity ID & UI components	Q-learning	Code	UI Automator
5	**AimDroid** (Gu et al., 2017)	Best first action	Activity ID & UI components	SARSA	Activity	Monkey & UI Automator

Vuong & Takada (Vuong & Takada, 2018) proposed a Q-Learning-based automated test case generation tool designed for Android apps using the Markov model to describe the AUT. The tool learns the most relevant behavioral model of the AUT and generates test cases based on this model. The tool executes a sequence of a fixed number of events, also called an episode. After finishing an episode, the tool selects a random state from those that have already been visited and starts a new episode in the next phase. However, this tool has multiple limitations. For example, it only generates UI events and does not cover activities triggered by system events.

Adamo, Khan, Koppula, & Bryce (Adamo, Khan, et al., 2018) introduced a Q-Learning-based automated test case generation tool designed for Android apps built on the top of Appium and UI Automator. During the process of test case generation, the tool chooses an event with the highest Q-value from the set of available events in each state. The test case generation process is quite similar to previous work (Vuong & Takada, 2018), and this generation process is divided into episodes where the states used in previous episodes are employed as a basis for beginning a new episode. The authors define the state to be the set containing the unique actions available.

QBE (Koroglu et al., 2018) Q-Learning-based exploration approach generates test cases. Instead of using a random exploration approach, the GUI is explored based on a pre-approximated probability distribution that satisfied a test objective. It creates a Q-matrix that shows the probabilities of reaching the test objective, which is used to select the next action. However, QBE 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.

AimDroid (Gu et al., 2017) is a model-based test case generation tool for Android apps. AimDroid implements an RL-guided random approach. AimDroid is composed of two activities: it runs a breadth-first search to discover unexplored activities and insulates the discovered activity in a "cage" and intensively exploits such activity using RL-guided fuzzing algorithms. This tool divides the tests into episodes; each episode generates a bounded number of events and focuses on a single activity by disabling activity transitions. Furthermore, AimDroid uses an RL algorithm called SARSA to learn about the capability of events that can explore new activities, to "look ahead" and to select events that are more likely to trigger new activities and crash greedily. AimDroid also has some limitations. For example, it disables the activity transition, which may drop some faults caused by the Activity life cycle. Moreover, AimDroid does not learn the second-best event to choose from, it only knows the best SARSA-based event, and for all other events, it chooses randomly.

Humanoid (Li et al., 2019) was implemented along with Droidbot (Li et al., 2017) 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. As a limitation, this tool does not present an increment in coverage when compared to other tools. It is unable to use textual information available in the app to generate test cases.

4.3 Justification of the Proposed Approach

There are several techniques of reinforcement learning explained in section 4.1, Q-Learning is the most suitable for GUI testing, among other reinforcement learning techniques, since other techniques require many actions to be generated during the learning process, which is costly (Koroglu et al., 2018). The idea behind using Q-Learning is that the tabular Q-function is rewarded with each selection of possible actions over the app. However, this reward may vary according to the test objective. Thus, events that are never selected can present a higher reward than events that have already been executed, which reduces the redundant execution of events and increases coverage.

Q-Learning has been used in software testing in the past and has shown better results to improve the random exploration strategy (Adamo, Khan, et al., 2018; Koroglu et al., 2018; Mariani et al., 2011; Vuong & Takada, 2018). However, a common limitation to all these tools is that the reward function assigns the highest reward when the event is executed for the first time to maximize coverage or locate crashes. Nonetheless, in the proposed approach, the environment does not offer direct rewards to the agent. The agent itself tries to visit all states to collect more rewards. The proposed approach uses tabular

Q-Learning like other approaches but uses an effective exploration strategy that reduces actions redundant execution and uses different states and action spaces. Action selection is the main part of Q-Learning in finding an optimal policy. The policy is a process that decides on the next action a from the set of current actions. Unlike previous studies, the proposed approach utilizes the upper confidence bound (UCB) exploration-exploitation strategy as a learning policy to create an efficient exploration strategy for GUI testing. UCB tries to ensure that each action is explored well and is the most widely used solution for multi-armed bandit problems (Lonza, 2019). The UCB strategy is based on the principle of optimism in the face of uncertainty.

4.4 Implementation of the Proposed Approach

Q-Learning technique with UCB exploration strategy was adopted to generate a GUI test case for Android apps to improve coverage and crash detection. This approach was built in a test tool named DroidbotX. Moreover, the main idea of using DroidbotX was to evaluate the practical usefulness and applicability of the proposed approach. DroidbotX works with Droidbot (Li et al., 2017). Droidbot is a UI-guided input generation tool used mainly for malware detection and compatibility testing. Droidbot was chosen because it is open-source and can test apps without having access to the apps' source code. Moreover, it can be used on an emulator or real device without instrumentation and is compatible with all Android APIs.

The DroidbotX algorithm tries to visit all states because it assumes "optimism in the face of uncertainty". The principle of optimism in the face of uncertainty is known as a heuristic in sequential decision-making problems, which is a common point in exploration methods. The agent believes that it can obtain more rewards by reaching the unexplored parts of the state's space (Kamiura & Sano, 2017). In this principle, actions are selected greedily, but strong optimistic prior beliefs are put on their payoffs so that strong contrary evidence is needed to eliminate the action from consideration. This technique has been used in several RL algorithms, including the interval exploration method (Sutton & Barto, 1998). In other words, it means that visiting new states and making new actions would bring the agent more reward than visiting old states and making old actions. Therefore, it starts from an empty Q-function matrix and assumes that every state and action reward an agent with $+1$. When it visits the state s and makes an action a, the Q-function $Q\ (s, a)$ decreases, and the priority of the action a for the state s becomes lower. Our DroidbotX approach generates sequences of test inputs for Android apps that do not have an existing GUI model. The overall DroidbotX architecture is shown in Figure 4.2.

In Figure 4.2, the adapter acts as a bridge between the test environment and the test generation algorithm. The adapter is connected to an Android device or an emulator via the Android Debug Bridge (ADB). The adapter observer monitors the AUT and sends the current state to the test generator. Simultaneously, the executor receives the test inputs generated by the algorithm and translates them to commands. Furthermore, the test generator interacts and explores the app's functionalities following the observe-select-execute strategy, where all the GUI actions of the current state of AUT are observed; one action is selected based on the selection strategy under consideration, and the selected action is executed on the AUT. Similar to other test generators, DroidbotX uses a GUI model to save the memory of transitions called a UI transition graph (UTG). The UTG guides the tool to navigate between the explored UI states. The UTG is dynamically constructed at runtime, which is a directed graph whose nodes are UI states, and the edges between the two nodes are actions that lead to UI state transitions. The state node contains the GUI information and the running process information, and the methods are triggered by the action. DroidbotX uses Q-Learning-based test coverage approach shown in algorithm 1 and constructs a UI transition graph in algorithm 2.

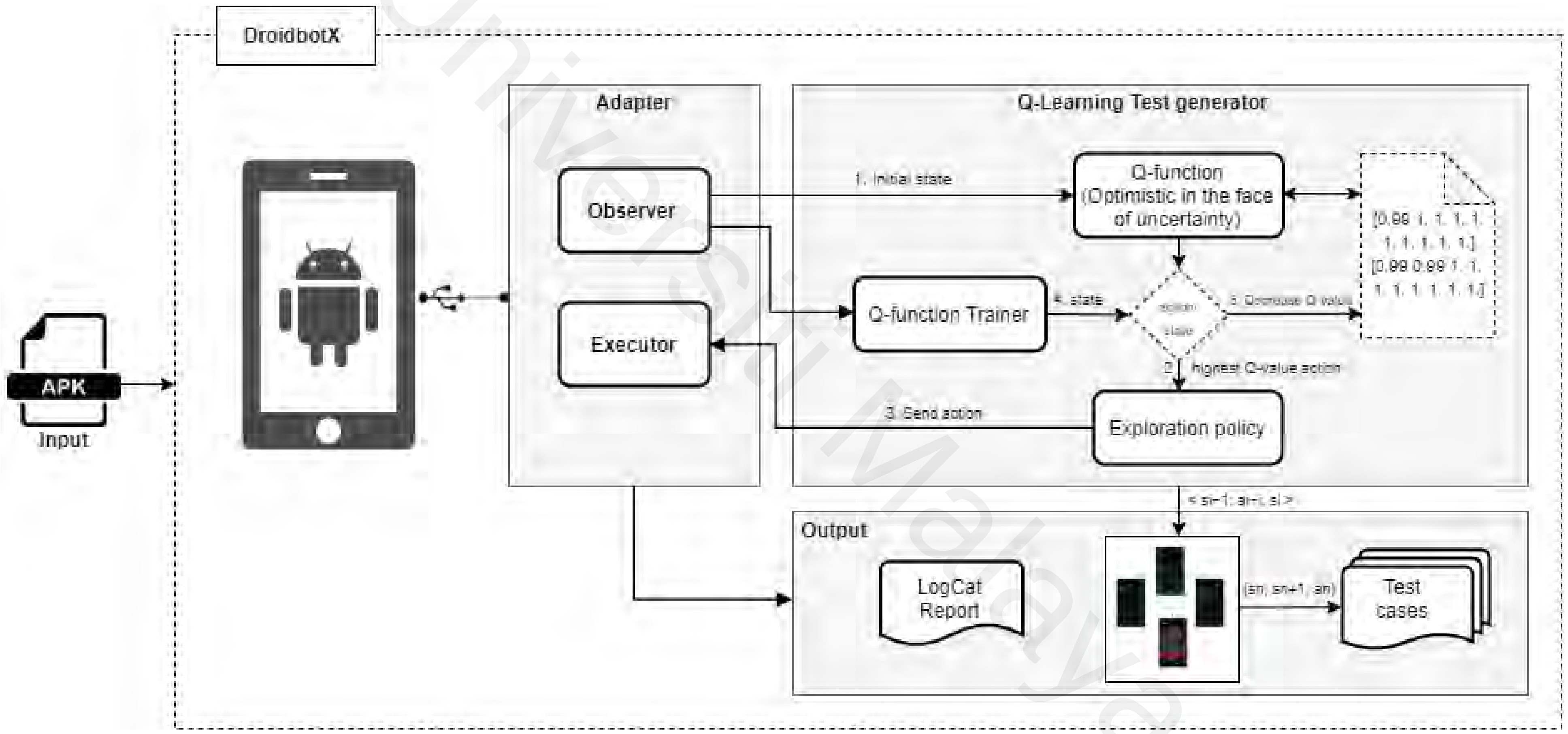


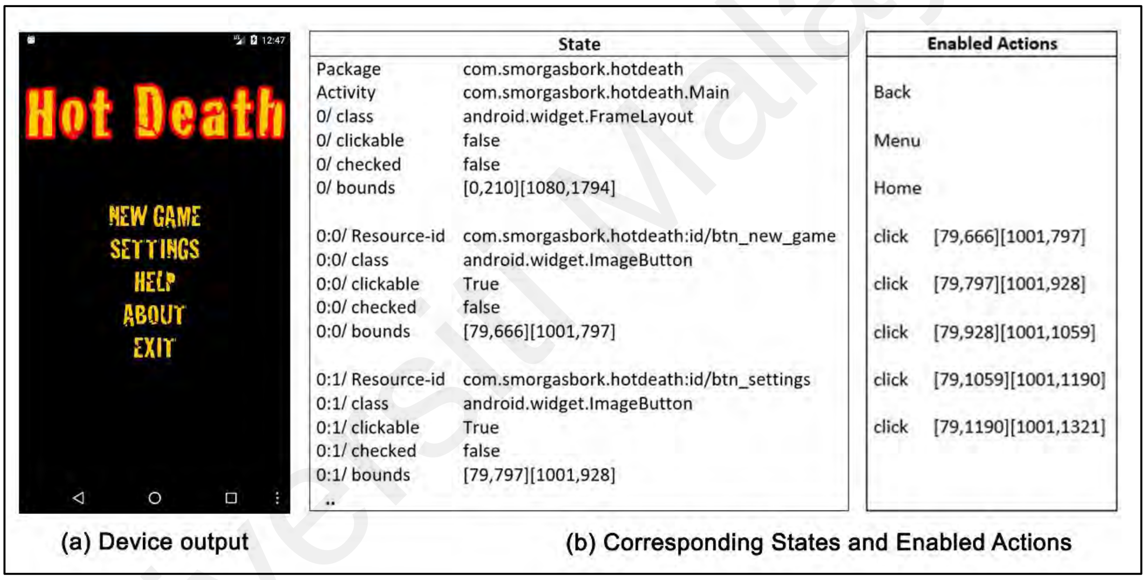
Figure 4.2: Overview of DroidbotX

4.4.1 States and Action Representation

In the Android app, all the UI widgets of an app activity are organized in a GUI view tree (Baek & Bae, 2016). The GUI tree can be extracted via UI Automator, which is a tool provided by the Android SDK. UI widgets include buttons, text boxes, search bars, switches, and number pickers. Users interact with the app using click, long-click, scroll, swipe up, swipe down, input text, and other gestures collectively called as GUI actions or actions. Every action is represented by its action type and target location coordinates. The GUI action is either (1) widget-dependent such as click and text, or (2) widget-independent such as the back that presses the hardware back button. A 5-tuple denotes an action: $a = (w, t, v, k, i)$, where w is a widget on a particular state, t is a type of action that can be performed on the widget (e.g., click, scroll, swipe), and v holds arbitrary text if widget w is a text field. For all non-text field widgets, the v value is empty. Moreover, k is the key event that includes back, menu, and home buttons on the device, and i is a widget ID. Note that DroidbotX sends an intent action that installs, uninstalls, and restarts the app.

State abstraction refers to the procedure that identifies equivalent states. In this approach, state abstraction determines two states as equivalent if (1) they have similar GUI content which includes package, activity, widget's type, position, and widgets parent-child relationship, and (2) they have the same set of actions on all interactive widgets, which is widely used in previous GUI testing techniques (Bauersfeld & Vos, 2014; Choi et al., 2013; Gu et al., 2017). GUI state or state $s \in S$ describes the attributes of the current screen out of the Android device where S denotes the set of all states. A content-based comparison and a set of actions to decide state equivalence, where two states with different UI contents and different enabled actions are assumed to be different states.

For simplifying states and actions representation, take an example of the Hot death app. Hot death is a variation of the classic card game. The main page includes a new game, settings, help, about, and exit buttons. Figure 4.3 shows a screenshot of the app's main activity, initial state, and related widgets with a set of enabled actions. Widget-dependent action is detected when a related widget exists on the screen. For example, a click-action exists only if there is a related widget with the attribute clickable true. Widget-independent action is available in all states because the user can press on device hardware buttons such as the home all the time.



State	
Package	com.smorgasbork.hotdeath
Activity	com.smorgasbork.hotdeath.Main
0/ class	android.widget.FrameLayout
0/ clickable	false
0/ checked	false
0/ bounds	[0,210][1080,1794]
0:0/ Resource-id	com.smorgasbork.hotdeath:id/btn_new_game
0:0/ class	android.widget.ImageButton
0:0/ clickable	True
0:0/ checked	false
0:0/ bounds	[79,666][1001,797]
0:1/ Resource-id	com.smorgasbork.hotdeath:id/btn_settings
0:1/ class	android.widget.ImageButton
0:1/ clickable	True
0:1/ checked	false
0:1/ bounds	[79,797][1001,928]
..	

Enabled Actions	
Back	
Menu	
Home	
click	[79,666][1001,797]
click	[79,797][1001,928]
click	[79,928][1001,1059]
click	[79,1059][1001,1190]
click	[79,1190][1001,1321]

Figure 4.3: An example of state and actions representation from Android app

4.4.2 Exploration Strategy

Android apps can have complex interactions between the events that can be triggered by UI widgets, and states that can be reached, and the resulting coverage achieved. In automated testing, the test generator must choose not only which widget to interact with, but also what type of action to perform. Each type of action on each widget is likely to improve coverage. Our goal is to interact with the app's widgets by sending relevant actions for each widget dynamically. This reduces the number of ineffective actions performed and explores as much app state as possible. Thus, UCB was used as an exploration policy to explore the app for new states and try out new actions. For each state, all potential widgets are extracted with their IDs and location coordinates, and then systematically choose be-tween five different actions (i.e., click, long-click, scroll, swipe left/right/up/down, and in-put text data) to interact with each widget. Next, whether the action brings the app to a new state by comparing its contents with all other states in the state model. If the agent identifies a new state, the exploring policy on the new state is recursively applied to discover unexplored actions. The exploration policy does not know about the consequences of each action, and the decision is made based on the Q-function. When exploration of this state terminates, the intent was executed to restart the AUT. Android intent is the message that passed between Android app components such as the start activity method to invoke activity. Examples of termination, an action that cause the AUT to crash, an action that switches to another app, or a clicks home button. The home action always closes the AUT, while the back action often closes the AUT. The exploration passes the login screen by searching in a set of pre-defined inputs. Some existing tools such as Android Monkey will stop at the login screen, failing to exercise the app beyond the login page.

4.4.2.1 Observer and Rewarder

The goal of the observer is to monitor the results of actions on the AUT. The Q-function then rewards the actions based on the results. Algorithm 1 uses the input parameters to explore the GUI and produces a set of event sequences as a test case for AUT. The Q-function $Q(s, a)$ takes state s and action a. The Q-function matrix is constructed based on the current state. Each row in the matrix represents the expected Q-values for a particular state. The row size is equal to the number of possible actions for the state. The $getEventfromActor$ function at lines 23–26 obtains all the GUI actions of the current state of AUT. The actions' initial values on the current state are assigned as 1 at line 26. The $UpdateQFunction$ function at lines 13–21 decreases the value of the action to 0.99 when the test generator conducts this action in the state. When all action value is 0.99, the maximum value becomes 0.99, and the test generator starts to choose some actions again. Then one action is selected and executed, and when a new state is found, the Q-function trainer receives the next state and updates the Q-function matrix to the previous state. The test generator sends KeyEvents such as back button at lines 27–28, if the state is the last or if there are no new actions in the current state.

$$Q(S, A) = Q(S, A) + a * (\gamma * maxQ(S', a) - Q(S, A)) \qquad (4.3)$$

The Q-Learning algorithm uses equation (4.3) to estimate the value of $Q(s, \mathrm{a})$ iteratively. The Q-function is initialized by a default value. Whenever an agent executes an action a from state s to reach s' and receives a reward $r + 1$, the Q-function is updated as equation (4.3) where α is a learning rate parameter between 0 and 1 and γ is a discount rate.

4.4.2.2 Action Selector

Action selection strategy is a crucial feature of DroidbotX. Right actions can improve the likelihood and decrease the time necessary to navigate to various app execution states. In the initial state, the test generator chooses the first action based on a randomized exploration policy to avoid the systematic handling of GUI layouts in each state. Then, the test generator selects actions from the new states and generates event sequences in a way that attempts to visit all states. The Q-function calculates the expected future rewards for actions based on the set of states it visited. In each state, the test generator chooses an action that has the highest expected Q-value from the set of available actions using $getSoftArgmaxAction$ function at lines 32–36, then the predicted Q-value for that action is reduced. Therefore, the test generator will not choose it again until all other actions have been tried. Formalizing this mathematically, the selected action is picked by Equation (4.4).

$$\boldsymbol{action} = \underset{\boldsymbol{a}}{\text{argmax}} \left[\boldsymbol{Q_t}(\boldsymbol{s_t}, \boldsymbol{a^i}) + \sqrt[c]{\frac{\boldsymbol{IogN_{s_t}}}{\boldsymbol{N}(\boldsymbol{s_t}, \boldsymbol{a^i})}} \right] \tag{4.4}$$

Equation (4.4) depicts the basic idea of UCB strategy, the expected overall reward of action a is $Q_t(s_t, a^i)$, $IogN_{s_t}$ denotes how often action has been selected in s_t, while $N(s_t, a^i)$ is the number of times the action a^i was selected in state s_t, and c is a confidence value that controls the level of exploration (set to 1). This method is known as "exploration through optimism," and it gives less-explored action a higher value and encourages the test generator to select them. The test generator uses the Q-function learned by equation (4.3) and UCB strategy to select each action intelligently, which balances the exploration and the exploitation of AUT.

Algorithm 1: Q-learning based Test Generation

```
Input:    A, Application under test
Output:  S, set of states;
          Q, q-function for all the state-action pairs;
          P, transition matrix, epsilon;
          KeyEvent-exploration parameter

1       (S, Q, P) ← (Ø, Ø, Ø)
2       launch(A)
3
4       while true do
5             Event ← getEventfromActor(Q)
6             Update P[old_state, new_state, :] #adjusting P[old_state, new_state, Event]
7             Q← UpdateQFunction(Q, P)
8             Execute(Event)
9             If not enable:
10                  break
11      return (S, Q, P)
12
13      Function UpdateQFunction(Q, P)
14                Q_target ← (Ø)
15                for index in [0, 1, …, 9] do
16                        for s in S do
17                                Q_target[s] ← maximum of Q[s, event] for all events
18                        for s in S do
19                                for a in all events that was ever made do
20                                        Q[s, a] ← 0.99 * sum(Q_target[:] * P[s, :, a])
21                return Q
22
23      Function getEventfromActor(Q)
24                state ← getCurrentState()
25                if state is not in S:
26                        Q[state, :] ←1 # For all possible events from state
27                if RANDOM([0; 1]) < epsilon do
28                        event ←KeyEvent
29                else
30                        event ←getSoftArgmaxAction(Q[state])
31                return event
32
33     Function getSoftArgmaxAction(Q_state)
34                max_qvalue ← max(Q_state)
36                best_actions ← all events where Q_state[event] == max_qvalue
36                event  ← choose randomly from best_actions
37                return event
```

4.4.3 Test Case Generation

Test case TC is defined as a sequence of transitions. $TC = (s1, a1, s2), (s2, a2, s3), \dots, (sn, an, sn + 1)$, where n is the length of the test case. Each episode is considered to be a test case, and each test suite TS is a set of test cases. The transition is defined as a 3-tuple (start-state, ss; action, a; end-state, se). Algorithm 2 dynamically constructs a UI transition graph to navigate between the explored UI states. It takes three input parameters: (1) the app under test, (2) Q-function for all the state-action pairs generated by algorithm 1, and (3) test suite completion criterion. The criterion for test suite completion is a fixed number of event sequences (set to 1000). DroidbotX's test generator explores a new state si, and adds a new edge $< si - 1;\ ai - i;\ si >$ to the UI transition graph, where $si - 1$ is the last observed UI state and $ai - i$ is the action performed in $si - 1$. For instance, consider generation of a test suite for Hot death Android app. DroidbotX creates an empty UI transition graph G (line 1), explores the current state of AUT (line 3), observes all the GUI actions of the current state (line 5), and constructs a Q-function matrix. Then one action is selected and executed based on *getSoftArgmaxAction* function (line 7), when a new state is found, the *UpdateQFunction* function receives the next state and updates the Q-function matrix to the previous state. The transition of executed action, next state, and previous state are added to the graph (line 15). The Q-value of executed action is decreased to avoid using the same action of the current state. The process is repeated until completing the target number of actions. Figure 4.4 shows an example of UTG from the Hot death Android app.

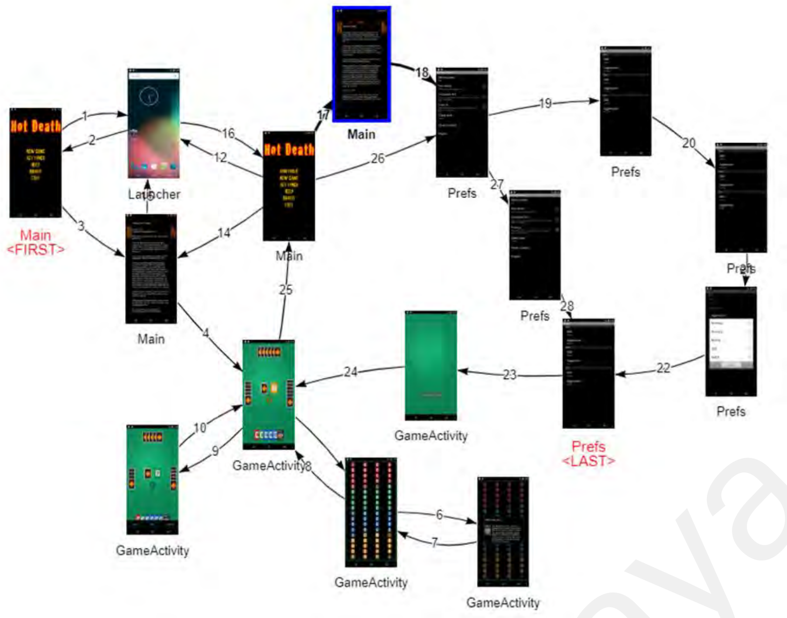


Figure 4.4: A UI state transition graph from a real-world Android app.

Algorithm 2: DroidbotX Test Suite Generation

Input: **AUT,** App under test
Input: **Q**, Q-function for all the state-action pairs
Input: **C**, Test suite completion criterion
Output: **TS**, Test Suite

```
1     Create an empty UI transition graph $G =< S, E >$
2     Run the AUT
3     Observe current UI state $s$ and add $s$ to $S$
4     repeat
5        Get All unexplored actions in $s$ as $A$
6        if $A$ is not empty then
7        Select as action $a$ from $A$ based on $Q(s)$
8     else
9     Extract a state $s$ in $S$ that has unexplored actions
10       Get the shortest path $p$ from $s$ to $s$ in $G$
11       Select the first action in $p$ as $a$
12    end if
13    Perform action $a$
14    Observe the new UI state $s$ new and add $s$ new to $S$
15    Add the edge <s, a, s new> to E
16    Until all actions in all states in $S$ have been explored
17    or
18    Until length of TS is equal $c$
19    end
```

4.5 Significance of the Proposed Approach

There are some significant features to the proposed approach that are discussed as follows.

- **Coverage:** The Q-Learning test coverage approach minimizes event sequences redundancy to maximize code coverage by using a Q-function matrix, where each action is executed once in each state.
- **Modeling:** The Q-Learning test coverage approach can explore more app functionalities and construct a more complete model without state explosion. The proposed approach uses two layers of state similarity and considers system events in the model.
- **Compatibility:** DroidbotX can be used on an emulator or real device without instrumentation and is compatible with all Android APIs. Compatibility testing is the capability of evaluating the app's correctness and robustness when running on different devices.
- **Test Artifacts:** DroidbotX generates log reports, activity coverage reports, and state transition models during the testing process.
- **Ease of use:** DroidbotX is easy to use because it is configurable and flexible enough to cater to various devices. Usability is a primary concern for tool developers; it affects reuse, research collaboration, and ultimately research impact.
- **Extendable:** DroidbotX integrated Gym, a toolkit for developing and comparing reinforcement learning algorithms. It further allows developers to customize the test exploration strategy using gym algorithms such as Deep Q Network, A2C, and SARSA.

4.6 Conclusion

This research aims to present a Q-Learning-based test coverage approach to generate GUI test cases for Android apps. This approach adopted a UCB exploration strategy to minimize redundant execution of events that improve coverage and crash detection. The proposed approach generates inputs that visit unexplored app states and uses the execution of the application on the generated inputs to construct a state-transition model generated during runtime. It visits all states in the face of uncertainty through the exploration of the new AUT states that generate new action that automatically produces more reward than visiting old states. This approach was implemented into the test tool named DroidbotX and it is publicly available.

CHAPTER 5: EVALUATION OF THE PROPOSED APPROACH

This chapter provides an evaluation of the DroidbotX approach using thirty Android apps collected from F-Droid. It is for this purpose that the empirical case study method was adopted. It was used in (Kitchenham et al., 2002; Perry et al., 2004) to analyze the effectiveness of DroidbotX. This comparison used four main metrics: instruction coverage, method coverage, activity coverage, and ability to detect crashes. In this chapter, the proposed approach's empirical evaluation compared to the existing frameworks and tools is also discussed. In the end, the evaluation results were compared with the results of the already existing tools.

In this chapter, the outline is organized as follows: Section 5.1 discusses the case study design. Section 5.2 describes the execution of the case study. The general discussion and statistical testing results are all presented in section 5.3. In section 5.4, the threats to validity are highlighted explicitly. Section 5.5 is the conclusion of the chapter.

5.1 Case Study Design

This evaluation adopts a similar case study method, just like the one described in section 3.1.

5.1.1 Research Questions for the Proposed Approach Evaluation

The major first step required in the empirical case study evaluation is to construct the research question that aligns and fulfills the evaluation purpose. Five questions were developed by embodying the evaluation objective:

RQ 1. What are the instructions, methods, and activity coverage achieved by DroidbotX compared to the state-of-the-art tools?

RQ.2. How effective is DroidbotX to detect unique app crashes compared to other state-of-the-art tools?

RQ.3. How does DroidbotX compare to the state-of-the-art tools in terms of test sequence length?

RQ.4. How effective is the model constructed by DroidbotX compared to the state-of-the-art tools?

RQ.5. What is the time complexity of DroidbotX algorithm?

5.1.2 Case Study Criteria

The effectiveness of the proposed approach compared to state-of-the-art tools was evaluated based on four criteria:

C1. Instruction Coverage (IC) refers to the Smali (Freke, 2013) code instructions through decompiling the APK installation package. It is the ratio of triggered instruction in the Java instruction code of the app to the total number of instructions. Huang et al. (2015) first proposed the concept of instruction coverage, which is used in many studies as an indicator to evaluate test efficiency (Choi et al., 2013; Gu et al., 2017; Gu et al., 2019; Koroglu et al., 2018). It is a more accurate and valid test coverage criterion that reflects the adequacy of testing results for closed-source apps (Yang et al., 2019).

C2. Method Coverage (MC) is the ratio of the number of methods called during the execution of the AUT to the total number of methods used in the source code of the app. By improving the method coverage, more functionalities of the app were explored and tested (Azim & Neamtiu, 2013; Choudhary et al., 2015; Dashevskyi et al., 2018).

C3. Activity Coverage (AC) is defined as the ratio of activities explored during execution to the total number of activities existing in the app. A high activity coverage value indicates that more screens have been explored and will therefore be more exhaustive for the app exploration.

C4. Crash detection: An Android app crashes when there is an unexpected exit caused by an unhandled exception. Crashes will result in the termination of the app's processes, and dialogue is displayed to notify the user about the app crash. The further code the tool explores, the more likely it is to discover potential crashes.

5.1.3 Apps Selection

For the experimental analysis, 30 Android apps were chosen from the F-Droid repository. These apps were earmarked from the repository based on two features:

1) the app's number of activities: the apps were categorized as small (number of activities less than five), medium (number of activities less than ten), and large (number of activities more than ten). The app’s activities were determined in the Android manifest file of the app.
2) app permissions required: In this study, only apps that require at least two of the permissions were selected to evaluate how tools react to different system events. These permissions include access to contacts, call logs, Bluetooth, Wi-Fi, location, and camera of the device. 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.

Table 5.1 lists the apps by app type, along with the package name, the number of activities, methods, and instructions in the app (which offers a rough estimate of the app size). Acvtool (Pilgun et al., 2020) was used to collect instruction coverage and method coverage. This tool does not require the source code of the app. Acvtool code coverage is based on Smali representation of the bytecode.

Table 5.1: Overview of Android apps selected for testing.

No	APP name	Package name	Version	Category	Instruction	Methods	Activity
1	Bubble	com.nkanaev.comics	4.1	Books	5208	463	2
2	WLAN Scanner	org.bitbatzen.wlanscanner	4	Communication	2484	141	1
3	Divide	com.khurana.apps.divideandconquer	2.1	Education	2306	195	2
4	Raele.concurseiro	raele.concurseiro	3	Education	1299	444	2
5	LolcatBuilder	com.android.lolcat	2.3	Entertainment	2497	79	1
6	MunchLife	info.bpace.munchlife	2.3	Entertainment	551	39	2
7	Currency	org.billthefarmer.currency	4	Finance	5461	148	5
8	Boogdroid	me.johnmh.boogdroid	4	Game	3984	398	3
9	Hot Death	com.smorgasbork.hotdeath	2.1	Game	17679	365	3
10	Resdicegame	com.ridgelineapps.resdicegame	1.5	Game	6853	144	4
11	Pushup Buddy	org.example.pushupbuddy	1.6	Health & Fitness	1985	165	7
12	Mirrored	de.homac.Mirrored	2.3	Magazines	3803	219	4
13	A2DP Volume	a2dp.Vol	2.3	Maps & Navigation	13452	600	8
14	Ethersynth	net.sf.ethersynth	2.1	Music & Audio	4056	168	8
15	Adsdroid	hu.vsza.adsdroid	2.3	Productivity	488	199	2
16	Talalarmo	trikita.talalarmo	4	Productivity	5122	658	3
17	Alarm Clock	com.angrydoughnuts.android.alarmclock	2.7	Productivity	5207	334	5
18	World Clock	ch.corten.aha.worldclock	2.3	Productivity	5200	315	4
19	Blockinger	org.blockinger.game	2.3	Puzzle	7090	356	6
20	Applications info	com.majeur.applicationsinfo	4.1	Tool	4806	315	6
21	Dew Point	de.hoffmannsgimmickstaupunkt	2.1	Tools	2282	75	3
22	drhoffmann	de.drhoffmannsoftware	1.6	Tools	5171	164	9
23	List my Apps	de.onyxbits.listmyapps	2.3	Tools	1930	96	4
24	Sensors2Pd	org.sensors2.pd	2.3	Tools	1346	149	4
25	Terminal Emulator	jackpal.androidterm	1.6	Tools	16098	994	8
26	Alogcat	org.jtb.alogcat	2.3	Tools	2344	199	3
27	Android Token	uk.co.bitethebullet.android.token	2.2	Tools	4658	288	6
28	Battery Circle	ch.blinkenlights.battery	1.5	Tools	963	79	1
29	Sensor readout	de.onyxbits.sensorreadout	2.3	Tools	994	683	3
30	Weather notifications	ru.gelin.android.weather.notification	2.3	Weather	8927	667	7

5.2 Case Study Execution

The experiments were executed on a 64-Bit Octa-Core machine with a 3.50 Gigahertz Intel Xeon® CPU running on Ubuntu 16.04 and 8 Gigabytes of RAM. The state-of-the-art GUI testing tools for Android apps were installed on the dedicated machine for running the experiments. Five tools with different techniques have been chosen for the experiment as follows Sapienz (K. Mao et al., 2016), Stoat (Su et al., 2017), Droidbot (Li et al., 2017), Humanoid (Li et al., 2019), and Android Monkey (Google, 2019j). These tools are the most recent techniques for Android test generation. Sapienz and Stoat have been adequately tested and are standard baselines in literature.

The Android emulator x86 ABI image (KVM powered) was used for the experiments. All comparative experiments ran on emulators because the publicly available version of Sapienz only supports emulators. In contrast, DroidbotX, Droidbot, Humanoid, Stoat, and Android Monkey support both emulators and real devices. Moreover, Sapienz and Stoat ran on the same version of Android 4.4.2 (Android KitKat, API level 19) because of their compatibility as described in previous studies (Choudhary et al., 2015; Gu et al., 2019); DroidbotX, Droidbot, Humanoid, and Android Monkey ran on Android 6.0.1 (Android Marshmallow, API level 23).

To achieve a fair comparison, a new Android emulator was used for each run to avoid any potential side-effects that may occur between the tools and apps. All tools were used with their default configurations. According to previous studies (Choudhary et al., 2015; Gu et al., 2019), Sapienz and Android Monkey were set to 200 milliseconds delay for GUI state updates. All testing tools were provided an hour to test each app, similar to other studies (Choudhary et al., 2015; Gu et al., 2017; K. Mao et al., 2016). To compensate for possible influence brought by randomness during testing, each test was repeated five times (with each test consisting of one testing tool and one applicable app being tested).

The final coverage and the progressive coverage were recorded after each action. Subsequently, the average value of the five tests was calculated as the final result.

5.3 Results and Discussion

In this section, the case study questions were answered by measuring and comparing four aspects: (i) instruction coverage, (ii) method coverage, (ii) activity coverage, and (iv) the number of detected crashes achieved by each testing tool on selected apps in our experiments. Table 5.2, Table 5.3, Table 5.4, and Table 5.5 show the results obtained from the six testing tools. The gray background cells in the tables indicate the maximum value achieved during the test. The percentage value is the rounded-up value obtained from the average of the five iterations of the tests performed on each AUT.

RQ.1: What are the instructions, methods, and activity coverage achieved by DroidbotX compared to the state-of-the-art tools?

The overall comparison results of the instruction coverage, method coverage, and activity coverage achieved by Android Monkey (M), Sapienz (Sa), Stoat (St), Droidbot (Dr), Humanoid (Hu), and DroidbotX (Q) on each subject Android apps are given in Table 5.2, Table 5.3 and Table 5.4.

1) Instruction coverage: The overall comparison of instruction coverage achieved by testing tools on selected Android apps is shown in Table 5.2. On average, DroidbotX achieves 51.5% instruction coverage, which is the highest across the compared tools. It achieved the highest value on 9 of 30 apps (including four ties, i.e., where DroidbotX covered the same number of instructions as another tool) compared to other tools. Sapienz achieved 48.1% followed by Android Monkey (46.8%), Humanoid (45.8%), Stoat (45%), and Droidbot (45%).

Table 5.2: Results on instruction coverage by test generation tools

Apps Under Test	Instruction coverage (%)					
	DroidbotX	**Droidbot**	**Humanoid**	**Sapienz**	**Stoat**	**Monkey**
Bubble	27.9	28.0	25.4	28.2	29.8	30.3
WLAN Scanner	59.4	59.2	58.6	61.3	57.2	58.9
Divide	59.1	56.5	57.4	57.4	55.4	60.8
Raele concurseiro	34.8	34.3	34.2	35.4	39.1	35.5
LolcatBuilder	24.9	24.5	24.6	23.2	21.6	23.2
MunchLife	75.1	72.8	76.5	75.0	75.7	73.6
Currency	49.0	38.6	45.4	49.1	50.3	47.0
Boogdroid	34.5	34.4	34.5	29.8	29.7	29.8
Hot Death	53.7	49.1	49.1	54.2	49.6	51.4
Resdicegame	72.9	66.2	61.6	71.5	64.9	67.2
Pushup Buddy	33.0	27.0	28.5	25.3	21.9	20.2
Mirrored	36.1	36.1	36.0	27.5	25.4	27.5
A2DP Volume	39.1	35.6	26.9	29.5	31.2	25.6
Ethersynth	82.7	47.9	64.8	77.4	55.7	71.5
Adsdroid	34.1	23.0	28.7	30.8	29.6	30.8
Applications info	68.3	57.7	45.7	45.7	38.9	29.4
Blockinger	69.3	67.0	67.9	66.4	66.5	66.0
Dew Point	72.8	67.0	72.9	71.9	68.9	68.2
drhoffmann	36.7	24.8	32.9	36.7	28.2	36.7
List my Apps	58.7	44.5	60.9	64.3	60.0	64.2
Sensors2Pd	70.4	71.1	74.2	73.6	71.3	74.1
Talalarmo	74.4	64.8	74.9	74.1	69.3	76.0
Terminal Emulator	40.9	35.6	36.2	41.4	34.1	40.0
Alarm Clock	64.1	66.3	61.8	49.1	48.5	59.6
Alogcat	70.1	50.6	51.1	62.7	74.8	56.7
Android Token	44.2	37.6	45.1	40.6	37.6	38.0
Battery Circle	81.3	74.0	81.3	78.1	78.1	77.6
Sensor readout	80.1	79.1	79.9	81.5	60.8	79.7
World Clock	46.1	24.8	33.8	41.3	42.5	44.4
Weather notifications	44.2	37.6	37.9	42.4	41.8	45.8
Overall average	**51.5**	**45.0**	**45.8**	**48.1**	**45.0**	**46.8**

Figure 5.1 presents the boxplots, where x indicates the mean of the final instruction coverage results across target apps. The boxes provide the minimum, mean, and maximum coverage achieved by the tools. It also shows that DroidbotX achieves the highest instruction coverage for all three app size groups. Better results from DroidbotX can be explained as it accurately identifies which parts of the app are inadequately explored. The DroidbotX approach is used to explore the UI components by checking all actions available in each state and avoiding the use of the explored action to maximize coverage. In comparison, Humanoid achieved a 45.8% average value and had the highest

coverage on 4 out of 30 apps due to its ability to prioritize critical UI components. Humanoid chooses from ten actions available in each state that are likely to interact with human users.

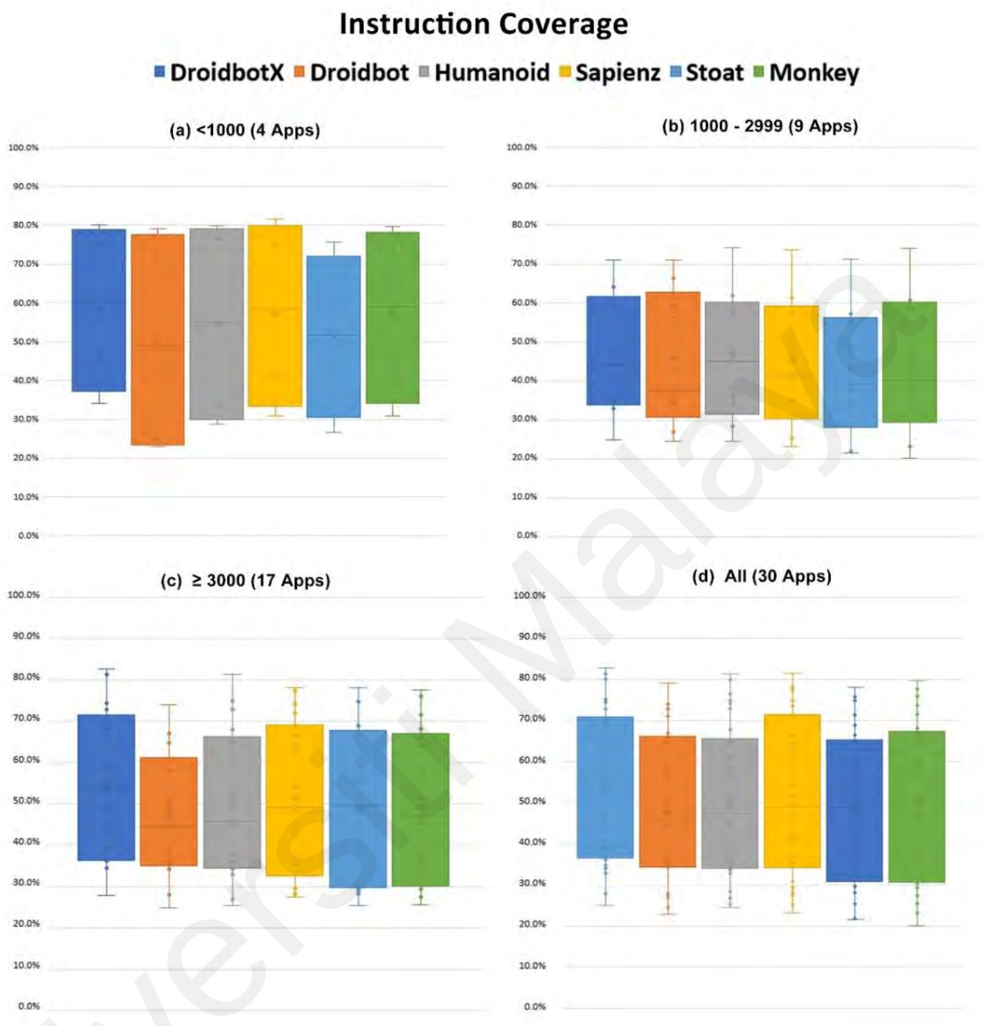


Figure 5.1: Variance of instruction coverage achieved across apps and five runs.

As seen in Figure 5.1, Android Monkey's coverage was close to Sapienz's coverage during a one-hour test. Sapienz uses Android Monkey to generate events and uses an optimized evolutionary algorithm to increase coverage. Stoat and Droidbot achieved lower coverage than the other four tools. Droidbot explores UI components in depth-first order. Although this greedy strategy can reach deep UI pages at the beginning, it may get stuck because the order of the event execution is fixed at runtime. Droidbot does not explicitly revisit the previously explored states, and this may fail to cover a new code that should be reached by different sequences.

2) Method coverage: DroidbotX significantly outperformed state-of-the-art tools in method coverage with an average value of 57%. The highest value on 9 out of 30 apps (including three ties where the tool covered the same method coverage as another tool). Table 5.3 shows that the coverage of app instructions obtained by the tools is lower than that of the method. This indicates that the method coverage cannot fully cover all the statements in the app's method. On average, Sapienz, Android Monkey, Humanoid, Stoat, and Droidbot achieved 53.7%, 52.1%, 51.2%, 50.9%, and 50.6% of method coverage, respectively. Stoat and Droidbot did not obtain the highest coverage of 50% on 10 of the 30 apps after five rounds of testing. In contrast, DroidbotX achieved the highest coverage of 50% in the twenty-four apps that were tested. In comparison, Android Monkey obtained less than 50% method coverage in eight apps. Sapienz displayed the best method coverage on 5 out of the 30 apps (including four ties where the tool covered the same method coverage as another tool). Sapienz's coverage was significantly higher for some apps such as "WLAN Scanner," "HotDeath," "ListMyApps," "SensorReadout," and "Terminal emulator". These apps have functionality that requires complex interactions with validated text input fields. Sapienz uses the Android Monkey input generation, which continuously generates events without waiting for the effect of the previous event. Moreover, Sapienz and Android Monkey can generate several events, broadcasts, and text that have not been supported by other tools. DroidbotX obtained the best results for several other apps, especially "A2DPVolume", "Blockinger", "Ethersynth", "Resdicegame", "Weather Notification", and "World Clock". DroidbotX approach assigns Q-values to encourage the execution of actions that lead to new or partially explored states. This enables the approach to repeatedly execute high-value action sequences and revisit the subset of GUI states that provides access to most of the AUT's functionality.

Table 5.3: Results on method coverage by test generation tools

Apps Under Test	Method coverage (%)					
	DroidbotX	**Droidbot**	**Humanoid**	**Sapienz**	**Stoat**	**Monkey**
Bubble	29.1	33.0	28.0	42.8	44.5	49.0
WLAN Scanner	65.5	65.4	64.4	66.0	59.3	63.5
Divide	63.9	54.5	54.5	52.8	47.2	71.7
Raele concurseiro	36.6	36.6	36.6	41.9	42.3	41.6
LolcatBuilder	32.9	32.9	32.9	32.9	27.8	29.9
MunchLife	66.2	66.2	66.7	66.7	66.7	62.1
Currency	58.1	40.4	53.8	58.8	61.5	55.8
Boogdroid	16.0	14.1	15.5	13.0	12.4	14.6
Hot Death	72.5	66.7	66.4	72.8	63.4	71.9
Resdicegame	66.8	60.6	53.6	62.4	51.3	52.6
Pushup Buddy	57.1	52.2	53.8	51.0	49.7	34.9
Mirrored	47.1	48.5	47.0	44.1	39.7	40.5
A2DP Volume	66.9	60.7	57.8	58.9	59.2	37.2
Ethersynth	85.2	61.8	68.1	78.5	67.1	68.8
Adsdroid	73.4	52.2	63.9	72.9	51.6	72.9
Applications info	65.1	61.6	53.8	58.8	51.8	40.5
Blockinger	82.0	79.1	79.2	77.3	78.5	62.5
Dew Point	75.7	74.9	75.7	75.7	73.9	61.9
drhoffmann	58.0	48.8	57.0	58.0	56.0	58.0
List my Apps	69.4	48.1	71.9	76.0	72.8	72.7
Sensors2Pd	80.7	81.6	90.6	81.9	83.2	86.6
Talalarmo	57.4	48.4	58.0	56.3	51.8	61.3
Terminal Emulator	53.9	52.2	51.2	54.4	46.2	50.7
Alarm Clock	79.9	81.0	66.8	49.7	50.2	52.8
Alogcat	80.9	73.2	75.5	78.9	92.2	69.4
Android Token	58.1	53.9	58.8	55.3	53.9	51.7
Battery Circle	83.5	76.5	85.6	84.1	83.5	83.5
Sensor readout	30.0	30.0	30.0	30.1	28.7	29.2
World Clock	57.5	39.5	39.1	43.8	54.9	54.5
Weather notifications	66.6	40.2	40.1	53.6	48.3	69.3
Overall average	**57.0**	**50.6**	**51.2**	**53.7**	**50.9**	**52.1**

Figure 5.2 presents the boxplots, where x indicates the mean of the final method coverage results across target apps. DroidbotX had the best performance compared to state-of-the-art tools, and Android Monkey was used for evaluation in most Android testing tools. Android Monkey can be considered a baseline because it comes with an Android SDK and is popular among developers. Android Monkey obtained a lower coverage comparing to DroidbotX because of its redundancy and random exploratory approach.

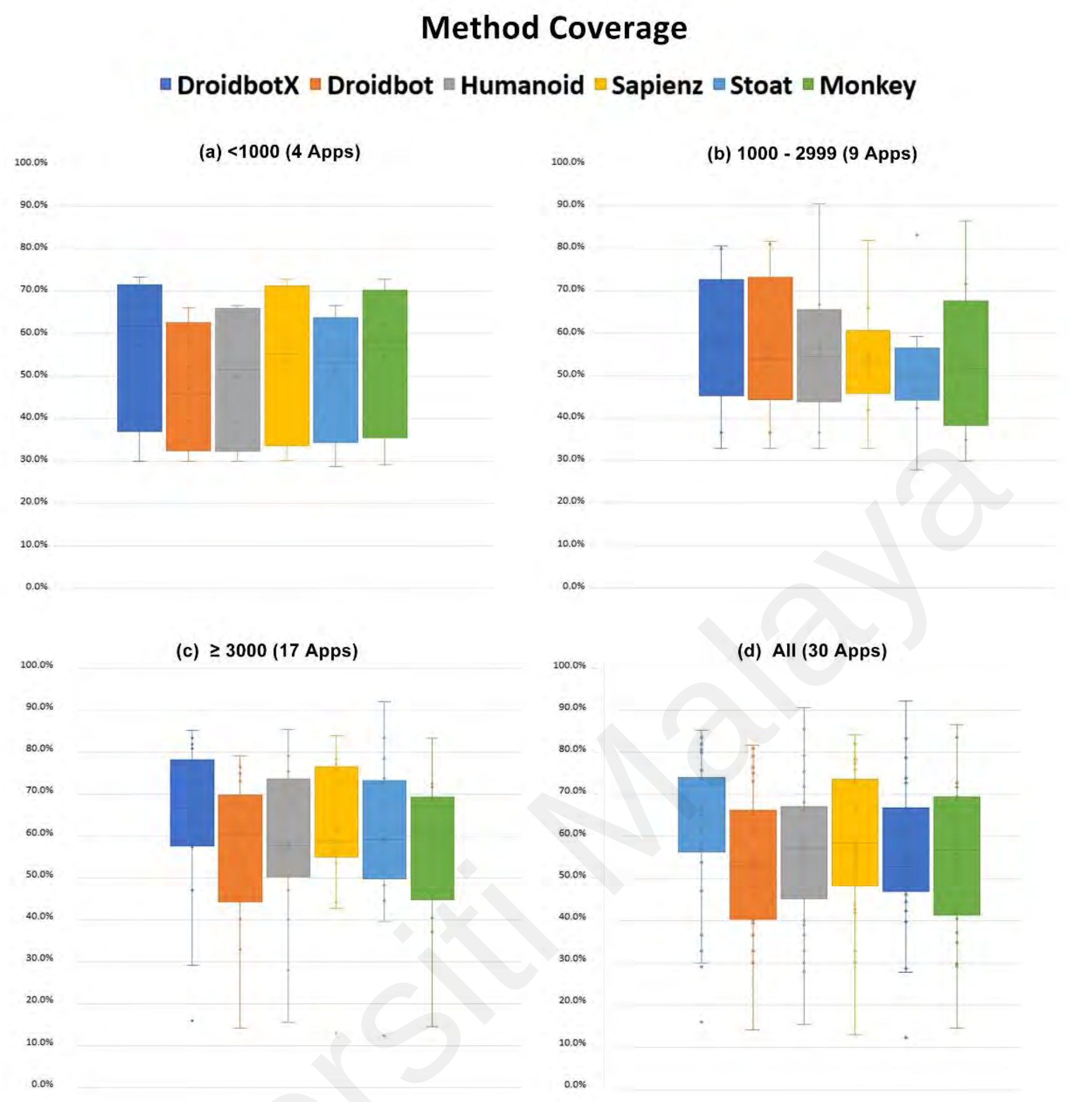


Figure 5.2: Variance of method coverage achieved across apps and five runs.

3) Activity coverage: The activity coverage is measured by intermittent observation of the activity stack on the AUT and recording all activities listed down in the android manifest file. The activity coverage metric was chosen because, once DroidbotX has reached an activity, it can explore most of the activity's actions. The results determine activity coverage differences between DroidbotX and other state-of-the-art tools. The resulting average value of the tools revealed that the activity coverage performed better than instruction and method coverage, as shown in Table 5.4.

Table 5.4: Results on activity coverage by test generation tools

Apps Under Test	Activity coverage (%)					
	DroidbotX	Droidbot	Humanoid	Sapienz	Stoat	Monkey
Bubble	50.0	50.0	50.0	50.0	50.0	50.0
WLAN Scanner	100	100	100	100	100	100
Divide	100	100	100	100	100	100
Raele concurseiro	100	100	100	100	100	100
LolcatBuilder	100	100	100	100	100	100
MunchLife	100	100	100	100	100	100
Currency	100	92.0	96.0	100	100	88.0
Boogdroid	86.7	66.7	80.0	100	40.0	46.7
Hot Death	100	100	100	100	100	73.3
Resdicegame	100	100	100	100	100	100
Pushup Buddy	71.4	68.6	71.4	62.9	62.9	62.9
Mirrored	75.0	75.0	75.0	75.0	75.0	85.0
A2DP Volume	95.0	90.0	85.0	100	100	97.5
Ethersynth	100	100	100	100	100	92.5
Adsdroid	100	100	100	100	100	100
Applications info	100	100	100	100	100	100
Blockinger	100	100	100	100	100	100
Dew Point	100	100	100	100	100	100
drhoffmann	97.8	91.1	93.3	93.3	95.6	86.7
List my Apps	100	55.0	100	100	100	100
Sensors2Pd	100	100	100	100	100	100
Talalarmo	100	100	100	100	100	100
Terminal Emulator	37.5	37.5	37.5	37.5	37.5	35.0
Alarm Clock	100	96.0	64.0	60.0	92.0	76.0
Alogcat	66.7	66.7	66.7	66.7	66.7	66.7
Android Token	66.7	53.3	60.0	66.7	50.0	66.7
Battery Circle	100	100	100	100	100	100
Sensor readout	66.7	66.7	66.7	66.7	66.7	66.7
World Clock	100	85.0	95.0	95.0	95.0	70.0
Weather notifications	57.1	57.1	57.1	45.7	42.9	37.1
Overall average	**86.5**	**82.1**	**83.3**	**84.0**	**83.0**	**80.0**

DroidbotX outperformed the other tools in its activity coverage, such as instruction and method coverage. DroidbotX has an average coverage of 86.5%, which was best achieved by the "Alarm Clock" app (including 28 ties, i.e., whereby DroidbotX covered the same number of activities as another tool). DroidbotX outperformed other tools because it did not explicitly revisit previously explored states due to its reward function. This was followed by Sapienz and Humanoid, with the average mean value of activity coverage at 84% and 83.3%, respectively. Stoat successfully outperformed Android Monkey in activity coverage with an average activity coverage of 83% due to an intrusive null intent fuzzing that can start an activity with empty intents. All tools under study were able to cover more than 50% of coverage on 25 apps, and four testing tools covered 100% activity coverage on 15 apps. Android Monkey, however, achieved less than 50% activity coverage of about three apps. Android Monkey achieved the least activity coverage with an average mean value of 80%.

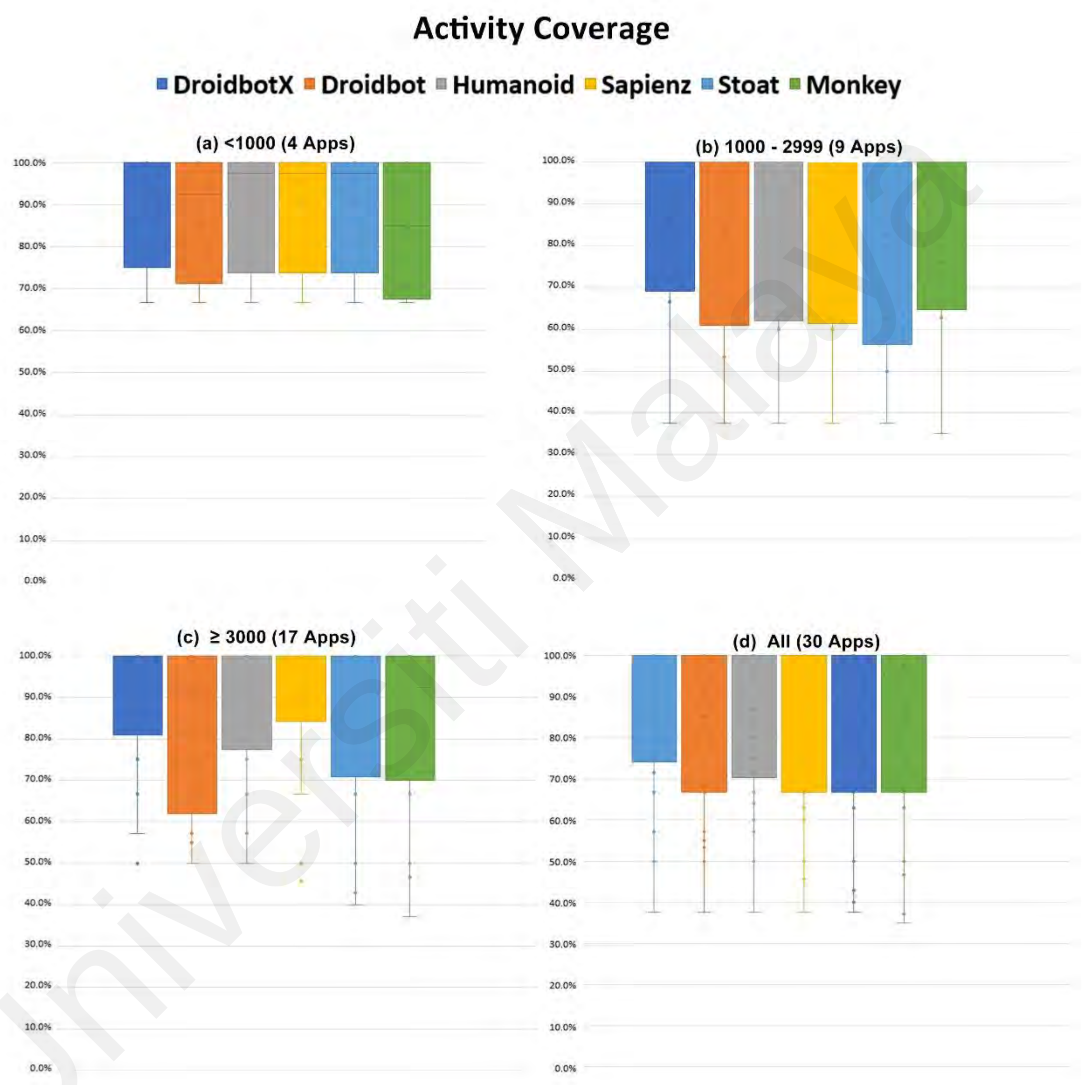


Figure 5.3: Variance of activity coverage achieved across apps and five runs.

Figure 5.3 shows the variance of the mean activity coverage of 5 runs across all 30 apps of the tool. The horizontal axis shows the tools used for the comparison. The vertical axis shows the percentage of activity coverage. Activity coverage was higher than the instruction and method coverage. DroidbotX, Droidbot, Humanoid, Sapienz, Stoat, and

Android Monkey obtained a 100% coverage increased from a mean coverage of 89%, 85%, 86.6%, 87.3%, 85.8%, and 83.4%, respectively. All tools were able to cover above 50% of the activity coverage. Although Android Monkey implemented more types of events than other tools, it achieved the least activity coverage. Android Monkey generates random events at random positions in the App activities. Therefore, its activity coverage can differ significantly from app to app and may be affected by the number of events sequences generated. To sum up, the high coverage of DroidbotX was mainly due to the ability of DroidbotX to perform a meaningful sequence of actions that could drive the app into new activities.

RQ.2: How effective is DroidbotX to detect unique app crashes compared to other state-of-the-art tools?

A crash is uniquely identified by the error message and the crashing activity. LogCat (Google, 2019c) is used to repeatedly check the crashes encountered during the AUT execution. LogCat is a tool that uses the command-line interface to dump logs of all the system-level messages. Log reports were manually analyzed to identify unique crashes from the error stack following Su et al. (Su et al., 2017) protocol. First, crashes unrelated to the app's execution by retaining only exceptions containing the app's package name and filter crashes of the tool itself, or initialization errors of the apps in the Android emulator. Second, compute a hash over the sanitized stack trace of the crash to identify unique crashes. Different crashes should have a different stack trace and thus a different hash. Each unique crash exception is recorded per tool, and the execution process is repeated five times to prevent randomness in the results. The number of unique app crashes is used as a measure of the performance of the crash detection tool. Crashes detected by tools on a different version of Android via normalized stack traces were not compared because different versions of Android have different framework code. In particular, Android 6.0 uses the ART runtime while Android 4.4 uses Dalvik VM, different runtime environments have different thread entry methods. Based on Figure 8, each of the tools compared complements the others in crash detection and has its advantages. DroidbotX triggered an average of 18 unique crashes in 14 apps, followed by Sapienz (16), Stoat (14), Droidbot (12), Humanoid (12), and Android Monkey (11). Like activity coverage, Android Monkey remains the same as it has the least capacity to detect crashes due to its exploratory approach that generates a lot of ineffective and redundant events.

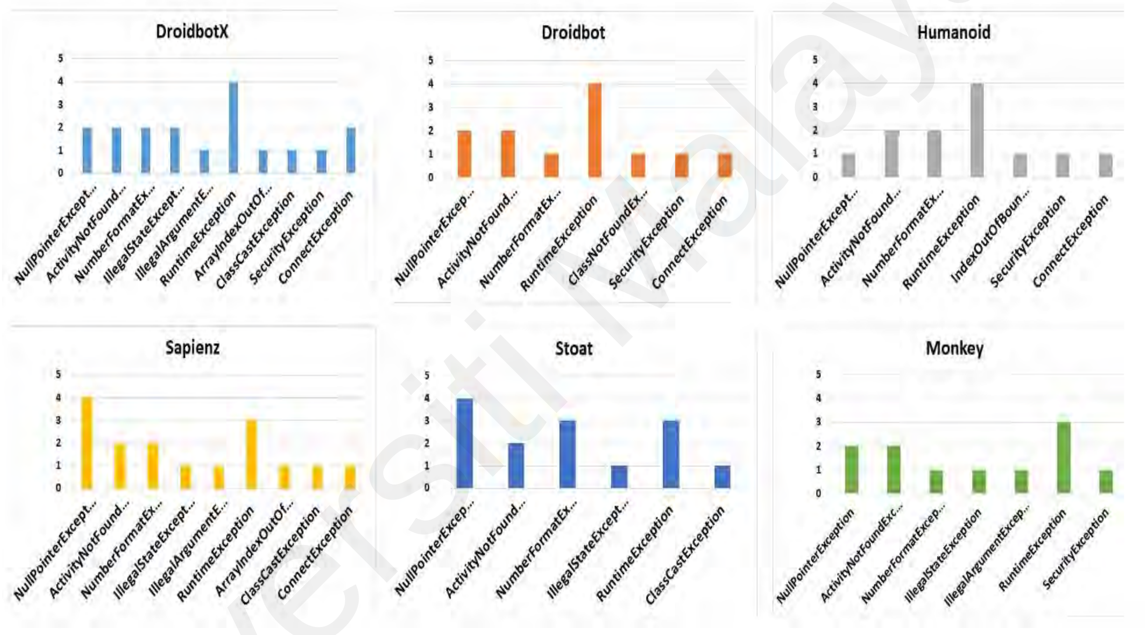


Figure 5.4: Distribution of crashes discovered.

Figure 5.4 summarizes the distribution of crashes by the six testing tools. Most of the bugs are caused by accessing null references. Common reasons are that developers forget to initialize references, access references that have been cleaned up, skip checks of null references, and fail to verify certain assumptions about the environments (Hu et al., 2014). DroidbotX is the only tool to detect IllegalArgumentException on the "World Clock" app, because it is capable to manage the exploration of states, and systematically sends back button events that may change the activity life cycle. This bug is caused by an incorrect redefinition of the onPause method of activity. Android apps may have incorrect behavior due to mismanagement of the activity's lifecycle. Sapienz uses Android Monkey to generate an initial population of event sequences (including both user and system events) prior to genetic optimization. This allows Sapienz to trigger other types of exceptions, including ArrayIndexOutOfBoundsException, and ClassCastException. For the "Alarm Clock" app, DroidbotX and Droidbot detected a crash on an activity that was not discovered by other tools in the five runs. Manually inspected several randomly selected crashes to confirm that they do appear in the original APK as well, and not found no discrepancy between the original and the instrumented APK behaviors.

Table 5.5: Statistics of crash results on apps by test generation tools understudy

Apps Under Test	# of Unique Crashes					
	DroidbotX	**Droidbot**	**Humanoid**	**Sapienz**	**Stoat**	**Monkey**
A2DP Volume	1	0	0	1	0	0
Alarm Clock	2	2	2	2	1	1
Alogcat	0	0	0	0	0	0
Adsdroid	1	1	1	1	1	1
Android Token	0	0	0	0	0	0
Applicationinfo	0	0	0	0	0	0
Battery Circle	0	0	0	0	0	0
Blockinger	0	0	0	0	0	0
Boogdroid	1	1	1	1	0	1
Bubble	1	1	1	1	1	1
Currency	0	0	0	1	1	1
Dew Point	2	1	1	2	2	1
Divide	0	0	0	0	0	0
Drhoffmann	2	1	2	2	2	2
Ethersynth	1	1	0	1	1	0
Hot Death	0	1	0	0	1	0
List my Apps	0	0	0	0	0	0
Lolcat Builder	1	1	1	0	0	1
Mirrored	0	0	0	1	0	1
MunchLife	0	0	0	0	0	0
Pushup Buddy	1	1	1	0	1	0
Raele.concurseiro	1	0	0	1	0	0
Resdicegame	0	0	0	0	0	0
Sensor readout	0	0	0	1	0	0
Sensors2Pd	1	1	1	1	1	1
Talalarmo	1	0	1	0	1	0
Terminal	1	0	0	0	1	0
Weather	0	0	0	0	0	0
World Clock	1	0	0	0	0	0
WLAN Scanner	0	0	0	0	0	0
Overall average	**18**	**12**	**12**	**16**	**14**	**11**

RQ.3: How does DroidbotX compare to the state-of-the-art tools in terms of test sequence length?

The cost of the proposed approach was measured as its running time and the number of inputs. First, the effectiveness of event sequence length on test coverage and crash detection was investigated. The event sequence length generally presents the number of steps required by the test generation tools to detect a crash. It is critical to highlight its effectiveness due to its significant effects on time, testing effort, and computational costs.

Table 5.6: Experimental results to answer case study questions.

Tools	Instruction Coverage (%)	Method Coverage (%)	Activity Coverage (%)	Number of crashes	Max Events Number
DroidbotX	51.5	57	86.5	18	1000
Droidbot	45	50.6	82.1	12	1000
Humanoid	45.8	51.2	83.3	12	1000
Sapienz	48.1	53.7	84	16	6000
Stoat	45	50.9	83	14	3000
Monkey	46.8	52.1	80	11	20,000

Table 5.6 shows that the Q-Learning approach implemented in DroidbotX achieved 51.5% instruction coverage, 57% method coverage, 86.5% activity coverage, and triggered 18 crashes within the shortest event sequence length compared to other tools. The results show that adapting Q-Learning with the UCB strategy can significantly improve the effectiveness of the generated test cases. DroidbotX generated a sequence length of 50 events per AUT state with an average of 623 events per run across all apps (which is smaller than the default maximum sequence length of Sapienz). DroidbotX completed exploration before reaching the maximum number of events (set to 1000) within the time limit. Sapienz produced 6,000 events and optimized events sequence

lengths through the generation of 500 events per AUT state. Nevertheless, it created the largest number of events after Android Monkey. However, the coverage improvement was closer to Humanoid and Droidbot, which generated a smaller number of events. Both Humanoid and Droidbot generated 1,000 events per hour. Sapienz uses Android Monkey that re-quires many events, which may include many redundant events to achieve high coverage. Hence, the coverage gained by Android Monkey only increases slightly as the number of events increases. Thus, a long events sequence length led to a minor positive effect on coverage and crash detection.

Second, the cost of the proposed approach was measured as its running time. Figure 5.5 depicts the progressive coverage of each tool over the threshold time used (i.e., 60 minutes). The progressive average coverage for all 30 apps was calculated every 10 minutes for each of the test generation tools in the study and a direct comparison of the final coverage was published. In the first 10 minutes, the coverage for all testing tools in-creased rapidly, as the apps had just started. At 30 minutes, DroidbotX achieved the highest coverage value compared to other tools. The reason is that the UCB exploration strategy implemented in DroidbotX finds events based on their reward and Q-value, which eventually tries to select and execute the previously unexecuted or less executed events, thus aiming for high coverage. Sapienz coverage increased rapidly, as the apps had just started, whereas all UI states were new but could not exceed the peak reached after 40 minutes. Sapienz has a high tendency to explore visited states, which could generate more event sequences. Stoat, Droidbot, and Humanoid had almost the same result and had better activity coverage than Android Monkey. Android Monkey could not exceed the peak reached after 50 minutes. The reason is that a random approach generates the same set of redundant events leading to a fall in its activity exploration ability. It is essential to highlight that these redundant events produced insignificant coverage improvement as the time budget increased.

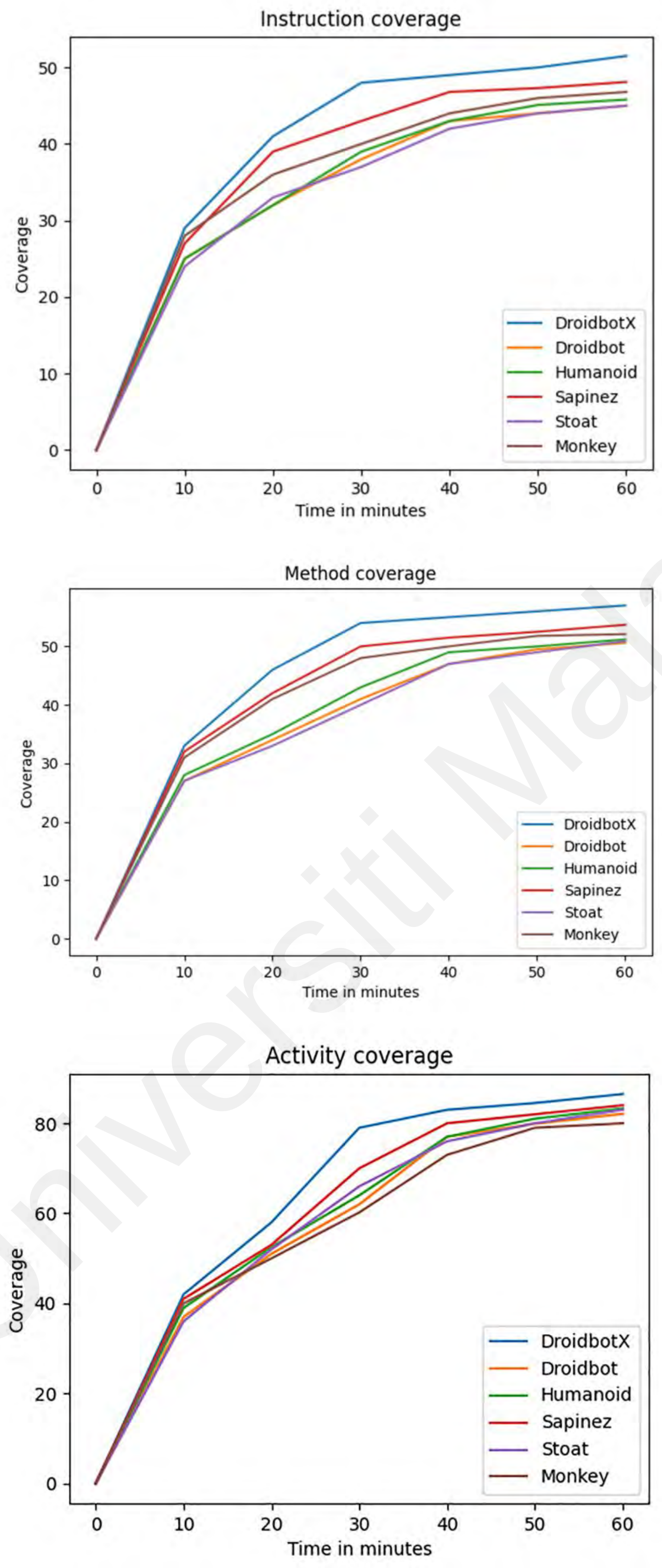


Figure 5.5: Progressive coverage

RQ.4. How effective is the model constructed by DroidbotX compared to the state-of-the-art tools?

The graph model enables DroidbotX to manage the exploration of states systematically to avoid being trapped in a certain state, which also can help to minimize unnecessary transitions. DroidbotX can achieve higher coverage than the other tools as indicated in Table 5.6, and its models are more compact without state explosion as shown in Table 5.7.

Table 5.7: Statistics of models built by Droidbot, Humanoid, and DroidbotX

	Droidbot			Humanoid			DroidbotX		
	Min	**Mean**	**Max**	**Min**	**Mean**	**Max**	**Min**	**Mean**	**Max**
Actions (#)	676	969	997	320	926	995	133	623	950
States (#)	9	69	306	8	60	304	10	75	304
Transition (#)	19	171	476	16	127	473	13	177	662

Table 5.7 shows the statistics of models built by Droidbot, Humanoid, and DroidbotX. These tools use the UI transition graph to save the memory of state transitions. DroidbotX generates an average of 623 events to construct the graph model, while Droidbot and Humanoid generate 969 and 926 average events, respectively. Droidbot cannot exhaustively explore app functions due to its simple exploration strategies. The depth-first systematic strategy used in Droidbot is surprisingly much less effective than the random strategy since it visits UIs in a fixed order and spends much time on restarting the app when no new UI components are found. Stoat requires more time for test execution due to its model construction in the initial phase which consumes time. Model-free tools such as Android Monkey and Sapienz can easily mislead exploration because of the lack of connectivity information between GUIs (Gu et al., 2019).

The results presented in Table 5.7 indicate that DroidbotX explores more app functionality and produces more comprehensive models. The evaluation results show that two-level state representation is appropriate to identify an app state. This state abstraction generates an acceptable number of states for an app, at the same time sufficiently captures functions.

RQ.5. What is the time complexity of DroidbotX algorithm?

In this section, the computation time complexity of the Q-learning test coverage algorithm was analyzed. Different Q-learning algorithms have different orders of complexity for the generation of a test case, ranging from the highest order of n^2, $n \log n$, to n, where n denotes the number of executed test cases. Intuitively, test case generation algorithms with higher complexity orders are supposed to have a stronger capability for fault detection. Normally, such an expectation exists, but not always (Anand et al., 2013).

Assuming the total number of events possible for the system is $O(E) = m$, and the total number of possible states is $O(S) = n$. Each system has its variable for computation so for average we will be taking computation time as $c, c1, k, k1, k2, k3, k4$ depending upon the system functionality and operations thrust. Table 5.8 shows Q-Learning-based test generation algorithm corresponds to the time it takes to run each line of code.

Table 5.8: Q-Learning-based test generation algorithm corresponds to time.

Q-learning based test generation	Computation cost/time per action
Input: **A**, Application under test	
Output: **S**, set of states;	
Q, q-function for all the state-action pairs;	
P, transition matrix, epsilon;	
KeyEvent-exploration parameter	
(S, Q, P) ← (Ø, Ø, Ø)	$k1$
launch(A)	$k2$
while true **do**	max number of times it will go through = $\boldsymbol{n}$
Event ← getEventFromActor(Q)	refer to the function, time= $\boldsymbol{c + n * k + c1 + k1 + 2 * n + k + c}$
Update P[old_state,new_state]	**P**(order of the matrix)
Q← UpdateQFunction(Q, P)	$\boldsymbol{k + 9(n[m]) + n([m] + [m])}$
Execute(Event)	$\boldsymbol{k3}$
If not enable:	$\boldsymbol{k4}$
break	$\boldsymbol{c}$
return (S, Q, P)	$\boldsymbol{c}$
Function UpdateQFunction(Q, P)	
Q_target ← (Ø)	$\boldsymbol{k}$
For index in[0,1,2..9] do	no of loops it will go through=**9**
For s in S do	no of loops it will traverse = $\boldsymbol{O(S) = n}$
Q__target[s]<-- maximum of Q[s,event] for all events	time = order of Event = $\boldsymbol{O(E) = m}$
For s in S do	time=order of states= $\boldsymbol{O(S) = n}$
For a in all events that was ever made do	no of loops to be iterated = $\boldsymbol{O(E)}$
q[s,a]<--0.99*sum(Q_target[:]*P[s,:,a])	time = $\boldsymbol{O(S) = n}$
return Q	$\boldsymbol{c}$
Function getEventFromActor(Q)	
state← getCurrentState()	$\boldsymbol{c}$
If state is not in S:	$\boldsymbol{O(S) = n}$
Q[state,:]<--1	**k**
If Random[0,1]<epsilon do	$\boldsymbol{c1}$
Key ← keyEvent	$\boldsymbol{k1}$
else	
Event ← getSoftArgmaxAction(Q_state)	$\boldsymbol{(O(S) + O(S) + k + c)}$
return event	
Function getSoftArgmaxAction(Q_state)	
max _qvalue← max(Q_state)	$\boldsymbol{O(S) = n}$
best_actions ← all events where Q_state[event]= maxqvalue	$\boldsymbol{O(S) = n}$
event ← choose randomly from best actions	$\boldsymbol{k}$
return event	$\boldsymbol{c}$

Assuming the total number of events possible for the system is m and the total number of possible states is n . Each system has its own variable for computation so for average we will be taking computation time as $c, c1, k, k1, k2, k3, k4$ depending upon the system functionality and operations thrust. Table 5.9 shows the action selector function corresponds to time.

Table 5.9: Action selector function corresponds to time.

Q-learning based test generation	Computation cost/time per action
While true **do**	max number of times it will go through = $\boldsymbol{n}$
Event ← getEventFromActor(Q)	refer to the function, time = $\boldsymbol{c + n * k + c1 + k1 + 2 * n + k + c}$
Update P[old_state,new_state]	**P**(order of the matrix)
Q← UpdateQFunction(Q, P)	$\boldsymbol{k} + \boldsymbol{9}(\boldsymbol{n}[\boldsymbol{m}]) + \boldsymbol{n}([\boldsymbol{m}] + [\boldsymbol{m}]))$
Execute(Event)	$\boldsymbol{k3}$
If not enable:	$\boldsymbol{k4}$
break	**c**
return (S, Q, P)	**c**

The total time for running the algorithm is calculated as follows

$$k1 + k2 + n[(c + n * k + c1 + k1 + 2 * n + k + c) + O[P] + k + 9(n[m]) + n([m] + [m])) + k3 + k4 + c)] + c$$

Assuming all constants to be average of c for the simplicity of the solution we have, the total time for running the algorithm is as follows

$$= 2 * c + n[\Sigma P + c * n + 2 * n + 4 * c + c + 9(n * m + 2 * n * m) + 3c] + c$$

$$= 2 * c + n[27n * m + c * n + 2 * n + 4 * c + 3 * c] + c$$

$$= 2 * c + n[27n * n * m + c * n + 2 * n + 7 * c] + c$$

$= K + n[27n * m]$

$= 27n * n * m + K$

$= O(27\,n * n * m)$, n and m being equivalent

for very high boundaries $= O(n * n * n) = O(n^3)$

For calculating the average time analysis, Table 5.10 shows Q-Learning-based test generation algorithm corresponds to time, we used the probabilistic distribution of the states and test cases and integrate them over time to find out the average value. As we have assumed, there is n number of states, probability of start state is $1/n$, probability of end state is $1/(n-1)$, probability of action between the transition of states $1/k$. Therefore, the probability of getting a test case is $1/n * 1/(n-1) * 1/k$. Let at any given point of time, there is r number of states in our matrix, therefore, the probability of the state not being in the matrix is $1 - r/n$. The average time for running the algorithm is calculated as follows $= n * [5c + n + r + K + 9[r * m/n + r * r\,a] + c\,]$. Taking all the constants to be equal to θ and the coefficient of $r * r$ be α , $= n * [n + \alpha * r.r + \theta]$ (5.1)

Where r is the range from 1 to n , therefore averaging $r * r$ over the sample

$= (n)(2n+1)(n+1)/6/n * n$ (5.2)

Using equation (5.2) in equation (5.1)

$= n * (n + n * \beta)$

$= O(n * n)$

Thus, the average running time is in order of $O(n^2)$.

Table 5.10: Q-Learning-based test generation algorithm corresponds to time for average time analysis.

Q-learning based test generation	Computation cost/time per action
Input: **A**, Application under test	
Output: **S**, set of states;	
Q, q-function for all the state-action pairs;	
P, transition matrix, epsilon;	
KeyEvent-exploration parameter	
(S, Q, P) ← (Ø, Ø, Ø)	$k1$
launch(A)	$k2$
while true **do**	max number of times it will go through $= n$
Event ← getEventFromActor(Q)	refer to the function, time= $5c + n + r$
Update P[old_state,new_state]	**P**(order of the matrix)
Q← UpdateQFunction(Q, P)	$K + 9(r * m/n + r * r\,a) + c$
Execute(Event)	$k3$
If not enable:	$k4$
break	c
return (S, Q, P)	c
Function UpdateQFunction(Q, P)	
Q_target ← (Ø)	k
For index in[0,1,2..9] do	no of loops it will go through=**9**
For s in S do	no of loops it will traverse= **O(S)=n*r/n**
Q__target[s]<-- maximum of Q[s,event] for all events	time = order of Event = $O(E) = m * r/n$
For s in S do	time=order of states= $O(S) = n * r/n$
For a in all events that was ever made do	no of loops to be iterated = $O(E) = n * r/n$
q[s,a]<--0.99*sum(Q_target[:]*P[s,:,a])	time =$O(S) = n * r/n$
return Q	c
Function getEventFromActor(Q)	
state← getCurrentState()	c
If state is not in S:	$O(S) = r/n * n$
Q[state,:]<--1	k
If Random[0,1]<epsilon do	$c1$
Key ← keyEvent	$k1$
else	
Event ← getSoftArgmaxAction(Q_state)	$(r/n * n + r/n * n + k + c)$
return event	
Function getSoftArgmaxAction(Q_state)	
max _qvalue← max(Q_state)	$O(S) = n * r/n$
best_actions ← all events where Q_state[event]= maxqvalue	$O(S) = n * r/n$
event ← choose randomly from best actions	k
return event	c

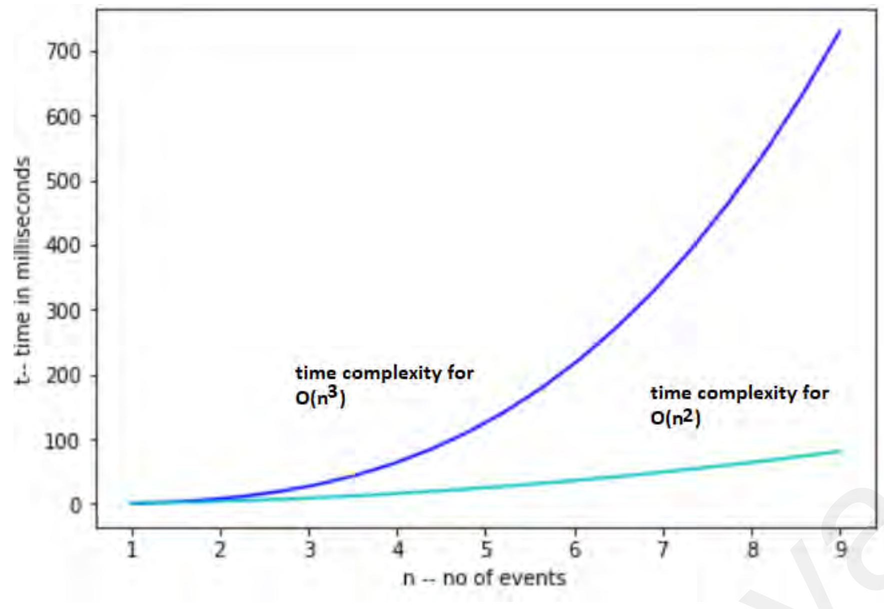


Figure 5.6: Time Complexity Analysis

Figure 5.6 represents the running time of the algorithm under worst-case scenarios and average case scenarios. The worst-case scenario $O(n^3)$ is the corresponding time for the execution when action, event, and states are all worst-case that is they take permutation of all the possible outcomes. While the average case scenarios $O(n^2)$, i.e., the time is calculated using the probability distribution of action, event, and states.

In a vast number of real-world problems, there are desired states to be reached through n-number of possible states by performing k number of actions. The initial matrix of the state space is unknown in such problems. Previous approaches for solving these kinds of problems required traversing all the possible paths or some of the paths were eliminated on some initial conditions, such types of solutions. The feasibility of using such solutions becomes very limited as the space size n increases took exponential time for the execution. In this research, however, it is shown that there are possible improvements in the structure and solutions of such problems. First, we can create a graph out of such problems and try to find out the shortest path, which will bring down the algorithm time analysis to $O(n^3)$, and then further we can use the probabilistic distribution of states and actions, considering some of the random combinations of states, action, and events, the test cases can be largely populated, and the time complexity is gradually reduced to $O(n^2)$. The same is proved in this research.

The complexity of the Q-learning test coverage algorithm is $O(n^3)$ when all the state, actions, and events are taken into account and these variables are accounted for the maximum number of occurrences, i.e., we assume that all these states, actions, and events will appear at some point in time. The complexity of the algorithm can be reduced significantly by randomly considering some of the events, states, and actions, we use some of the instances of the probabilistic distribution of states, actions, and events and average them over time. For instance, the probability of an event $= 1/n$ out of n events, probability of an action $= 1/k$ out of k action, and the probability of state $= 1/n$. Averaging the probabilities over time and using them in our equation, we find out that the average time of the algorithm reduces to $O(n^2)$.

Koenig and Simmons (1993) analyze the time complexity of the Q-Learning approach. They show that if the agent has initial knowledge of the state space or the state space has additional properties, the $O(n^3)$ bound can be decreased further. In the Q-Learning approach, the task of reaching a goal state for the first time is reduced from exponential time $O(en)$ to $O(n^3)$, by using the depth search method and finding out the shortest path between the source and the final destination, the complexity of the Q-Learning algorithm can be further decreased by considering duplicate values.

Our algorithm reduces the time complexity of $O(n^3)$ to $O(n^2)$ for the average case scenario as compared to the Q-Learning algorithm baseline (Koenig & Simmons, 1993). Here also we are using the depth search algorithm, each time a favorable state is reached we award the path matrix with the reward or incentive and comparing all the paths we come to our conclusion, but we randomize the probabilities and average them over time, this gives us randomized sets of outputs with optimized time complexity of $O(n^2)$.

5.4 Threats to Validity

There are threats and limitations to the validity of our study. Threats to internal validity, the non-deterministic approach of the tools results in obtaining different coverage for each run. Thus, multiple runs were executed to reduce this threat and to remove outliers that could affect the study critically. Each testing tool was allowed to run five times, and the test results were recorded and then computed to yield an average result of final coverage and progressive coverage of the tools. Another threat to the internal validity of our study is Acvtool's instrumentation effect, which affects the integrity of the results obtained. These may be caused by errors triggered by Acvtool's incorrect handling of the binary code or by errors in our experimental scripts. To mitigate this risk, the traces of our experiments for the subject apps were manually inspected.

External validity was threatened by the representativeness of the study to the real world. This means how closely the apps and tools were used in this study to reflect the real world. Moreover, the generalizability of the results was limited as we used a limited number of subject apps. To mitigate these, a standard set of subject apps was used in our experiment from various domains, including fitness, entertainment, and tools applications. The subject apps from F-Droid, which is commonly used in Android GUI testing studies, were carefully selected and the details of the selection process were explained in Section 5.1.3. Therefore, our test is not prone to selection bias.

5.5 Conclusion

This chapter provided an empirical evaluation of the effectiveness of DroidbotX and a comparison with GUI test generation tools for Android apps using 30 Android apps. Four criteria (i) instruction coverage, (ii) method coverage, (iii) activity coverage, and (iv) number of detected crashes, were set to evaluate and compare GUI test generation tools. Five tools with different techniques have been chosen for the experiment as follows Sapienz (search-based), Stoat (model-based), Droidbot (model-based), Humanoid (deep Q network), and Android Monkey (random-based). These tools are the most recent techniques for Android testing. Moreover, the computation time complexity of the Q-Learning test coverage algorithm was analyzed. The results showed that time complexity was reduced significantly from $O(n^3)$ to $O(n^2)$ for the average case scenario by randomly considering some of the events, states, and actions, and using the probabilistic distribution of states, actions, and events and average them over time. The experimental results reveal the capacity of the approach to achieve 51.5% instruction coverage, 57% method coverage, 86.5% activity coverage, and triggered 18 crashes within the shortest event sequence length over the five tools. The results show that the adaptation of Q-Learning with the UCB strategy can significantly improve the generated test cases' effectiveness. The study confirmed that the approach fulfilled its objective, and its aim was realized.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

From the presentation and evaluation of the proposed approach, it is evident that the proposed approach can efficiently and effectively maximize instruction coverage, method coverage, and activity coverage within the shortest event sequences and time limit. This chapter highlights the conclusions of this thesis and outlines the possible directions for future works. The chapter further explains how to fulfill this research's aim and objectives (in chapter 1). It presents this thesis's contributions and highlights the importance of the work carried out in this study. Lastly, the limitations of this study are highlighted, and the possible future works concluded.

In this chapter, the outline is organized as follows: Section 6.1 shows how this study's aim and objectives are attained, while section 6.2 describes this thesis's contributions. Section 6.3 highlights the significance of this work. This study's publications are listed in section 6.5, while the limitations and future works are shown in section 6.4.

6.1 Restatement of Research Objectives

This research aims to generate effective GUI test cases for Android apps that maximize coverage by minimizing redundant event execution. The full explanation of how to attain this research aims by realizing each objective which is stated in section 1.4.

- **To review the current state-of-the-art GUI testing tools to generate test cases for Android Apps.**

This research's objective was fulfilled, and the most credible works reported in articles and conferences were reviewed accordingly. The relatively new GUI testing tools for Android apps were comprehensively reviewed to analyze their strengths and weaknesses. A comprehensive thematic taxonomy was proposed based on an extensive review of Android apps' existing GUI testing tools. The critical features and related aspects of these tools are thoroughly examined to evolve the proposed taxonomy. According to the taxonomy parameters, the tools are exhaustively analyzed to explore shared traits and contrasts among existing tools. Finally, several research issues in Android app GUI testing are put forward that require further consideration to enhance Android app testing tools. This review is presented in chapter 2.

- **To examine the effectiveness of test generation tools for Android apps in terms of method coverage, activity coverage, and crash detection.**

To fulfill this objective, all the identified research problems (low coverage and events sequence length) were verified and addressed in this research. To analyze the significance of these research problems, an analytical-based analysis was carried out by using an empirical case study on 50 different Android apps downloaded from the F-Droid and AppBrain repositories. The test generation tools' effectiveness, especially in the events sequence length of the overall test coverage and crash detection, were evaluated against three criteria, method coverage, activity coverage, and many detected crashes. The findings indicate that a long events sequence performed better than the shorter events sequence. However, long events sequence led to a minor positive effect on the coverage and crash detection. Moreover, the results showed that the tools achieved less than 40% of the method coverage and 67% of the activity coverage. Furthermore, most of the tools find a fault in the user events, and none could find a fault in a system event. Besides, test generation tools generate text inputs randomly, which affects their coverage performance. This analysis is presented in chapter 3.

- **To develop an approach using Q-Learning to optimize test case generation that maximizes instruction coverage, method coverage, and activity coverage.** This study's objective was achieved by proposing an efficient test case generation approach that maximizes instruction, method, and activity coverage with minimizing redundant execution of events. The proposed solution is derived from the review between reinforcement learning and test case generation approaches to take advantage of both a randomly based approach and a model-based approach. Q-Learning technique with UCB exploration strategy was adopted to generate GUI test cases for Android apps to improve coverage and crash detection. This approach systematically selects events and guides the exploration to expose the functionalities of an AUT. It generates user and system inputs that target new states of the app and deploys the app's execution on the generated inputs to construct a state-transition model. Instead of randomly selecting the actions, the test generator learns how to explore new states by using new actions to gain more rewards. Thus, events never selected can present a higher reward than already executed events, which reduces the redundant execution of events and increase coverage. This approach was implemented into a test tool named DroidbotX. The tool was used to evaluate the practical usefulness and applicability of the approach. DroidbotX constructs a state-transition model of the app and generates test cases. These test cases follow the sequences of events that are the most likely to explore the app's functionalities. This was described in chapter 4.

- **To evaluate the ability of the proposed approach to generate effective test cases that detect crashes and maximize instruction coverage, method coverage, and activity coverage on real-world Android Apps.**

This objective was fulfilled by evaluating and analyzing the proposed approach using an empirical case study. The proposed approach was evaluated in comparison with top-quality test generation tools. DroidbotX was compared with Android Monkey, Sapienz, Stoat, Droidbot, Humanoid on 30 Android apps from the F-Droid repository. In the experiment, instruction coverage, method coverage, activity coverage, crash detection, running time, and test sequence length were analyzed to assess the approach's performance. All tools were tested and configured on a new emulator; each tool was run five times to avoid randomness during testing. The empirical data were validated by statistical analysis. Acvtool instrumented all the apps and collected the statistics of instruction and method coverage during testing. Acvtool does not require the source code of the app. Simultaneously, LogCat dumps a log of all the system-level messages and collects all fatal exceptions encountered during the AUT's execution for crash detection. The results show the significant performance of the proposed approach. The Q-Learning-based test coverage approach achieved (51.5%) instruction coverage, (57%) method coverage, (86.5%) activity coverage, and triggered (18) crashes within the time limit and shortest event sequence length compared to the other tools. The results show that Q-Learning adaptation with UCB exploration can significantly improve the generated test cases' effectiveness. The study confirmed that the approach fulfilled its objective, and its aim was realized.

6.2 Research Contributions

This study produces multiple contributions to the body of knowledge as follows.

- **Taxonomy of GUI testing tools for Android applications**

The taxonomy of GUI testing tools for Android applications was produced based on an extensive review of existing GUI testing tools for Android applications to analyze their strengths and weaknesses. The proposed taxonomy presents significant testing aspects such as automated activity testing, testing approach, type of evaluation method, and characteristics of the technologies used in state-of-the-art GUI testing tools. The taxonomy focused on test case generation as it is one of the most demanding testing activities tasks because of its strong impact on the whole testing process efficiency. The proposed taxonomy can serve as a basic tool to differentiate existing GUI testing tools. The taxonomy also helps researchers differentiate all GUI test case generation concepts for Android applications and explore the existing tools and techniques. The classification aims to be a guideline for researchers, testers, and test tool developers. This research's findings were presented in chapter 2.

- **Empirical Analysis of test generation tools for Android Applications**

The additional contribution has been inputted to the body of knowledge by empirically analyzing the effectiveness of six test generation tools for Android applications. The tools can generate user events and some of the system events that increase the possibility of finding faults on system events. Three criteria, methods coverage, activity coverage, and their ability to detect crashes were used to evaluate the testing tools. This study shows that longer events sequence led to a small positive effect on coverage and crash detection; Stoat and Android Monkey attained the highest number of events. Meanwhile, 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 incorporating other text input generation methodologies. This study appeared in chapter 3 and published in the literature in due time.

- **An effective approach for generating test for Android application based on Q-learning**

A novel approach was designed and implemented to overcome the shortcomings of the existing approaches lacking in the literature. The proposed solution is derived from the review between reinforcement learning, and test case generation approaches explained in chapter 2 to take advantage of both randomly based and model-based approaches. The proposed approach adopted the Q-Learning technique to generate an optimal GUI test case for Android applications to improve coverage, and crash detection. This approach was implemented into a test tool named DroidbotX and used the tool to evaluate the approach's practical usefulness and applicability. It generates user and system inputs that visit unexplored states of the app and uses the app's execution on the generated inputs to construct a state-transition model. Instead of randomly selecting the actions, the test generator learns how to act in an optimal way that explores new states by using new actions to gain more rewards. Thus, events never selected can present a higher reward than already executed events, which reduces the redundant execution of events and increase coverage. The proposed approach was highlighted in chapter 4 and made available in the literature for developers to use.

- **Empirical Analysis for the proposed approach**

The empirical and analytical evaluation techniques of the proposed approach were created using an empirical-based case study approach. The effectiveness of the proposed approach was performed and compared to the state-of-the-art tools. The approach was described in chapter 4, and the results are made available in chapter 5. Compared to the other approaches, the results' test reveals the capacity of the approach to achieve 51.5% instruction coverage, 57% method coverage, 86.5% activity coverage, and triggered 18 crashes within the shortest event sequence length over the five tools. The results show that the adaptation of Q-Learning with the UCB strategy can significantly improve the generated test cases' effectiveness. The study confirmed that the approach fulfilled its objective, and its aim was realized.

6.3 Significance of the Work

This section describes the significance of the proposed approach concerning the research problems found in section 3.4. The features are as follow

- Events sequence length: An important feature of the proposed approach is producing the shortest possible event sequence length while increasing the fault-finding probability. The proposed approach achieved the shortest event sequence by configuring the event inputs to 1000 to satisfy varying instruction, method, and activity coverage criteria and crashes detection. The configuration implemented revealed that the approach is practical for debugging and automatically reduces the testing space.

- Events sequence redundancy: The proposed approach generates test cases that are not redundant with no interaction. Test cases have similar steps where test cases may contain other tests or other tests with loops. A high redundancy negatively affects coverage efficiency since the testing tool has to generate many events to achieve the same coverage as one with low redundancy. Also, the capability to find crashes is reduced since the test suite tends to re-execute the same steps. The proposed approach generates effective GUI test cases. It uses an effective exploration strategy that reduces actions redundant execution. It tries to ensure that each action is well explored. The proposed approach reduces the number of ineffective actions performed and explores as much app state as possible by sending relevant actions for each widget.

6.4 Limitation and Future Work

Although DroidbotX has shown to be significantly better than existing tools for GUI test case generation of Android applications, there are several avenues of future research and improvement.

- Test Oracle: Currently, DroidbotX test coverage approach focuses on generating test cases, rather than constructing a test oracle that results in oracles that determine whether test cases pass or fail. Automated test oracle construction is a significant challenge beyond this research scope, but certainly as an avenue of future research interests. A test oracle is necessary to have the user in the loop to generate oracles that assess intended app behaviors. Hence, reducing the number of test cases to be inspected is certainly beneficial. To achieve fully automated testing of the Android app, the automated test generation technique alone is insufficient. A human tester must manually ascertain whether each test case diverges from the expectation. An automated test oracle is a technique used to prevent or solve this problem. Oracle is used to export test cases to executable test scripts during the implementation stage. A test case has been immediately materialized as a script; a human tester can initiate a test oracle simply by including assertions to the script.

- More improvement in coverage: DroidbotX can improve coverage significantly from state-of-the-art tools for GUI test case generation of Android applications, the test coverage is still relatively low. Specifically, the coverage is below 30% for some apps. This reason is that many apps need specific inputs such as login forms that are difficult or even impossible to generate automatically without being pre-programmed to behave optimally when confronting this form. The pre-programming approach cannot scale because different apps' login forms will appear different and function differently. The more effective approach is to provide an interface via which a human tester can pass knowledge about such special cases to an automated test generation tool. It is easy for a human tester to recognize a login form during testing and quickly pass it through by providing a proper credential. A possible solution is to incorporate human knowledge to offer the automatic tool's needed guidance with little effort. Moreover, the ability to generate tests that can achieve high code coverage has applications beyond testing for functional defects: Energy issues, latent malware, and portability problems are important concerns in the context of mobile devices that are often effectively detected by executing the code. This study's future work is to modify the approach to increase the coverage capacity and detect crashes and exploit the poor action discovered in test case generation to improve crash detection.

- Support more forms of inputs: Some forms of inputs, like system broadcasts, and sensor events, were not considered in our approach. This is a limitation that is peculiar to DroidbotX because the inputs require a special event generation strategy. However, it is not much of a huge problem at the moment. Most of the apps can be well-tested without involving these actions. Although DroidbotX does not predict the text when sending text input actions, there is the possibility that it could be fixed in the future by extending the model to support text prediction. Alternatively, it can be solved by integrating other text input generation techniques. Moreover, the model constructed by DroidbotX is still not complete since it cannot capture all possible behaviors during exploration, which is still an important research goal on GUI testing. All the events would introduce nondeterministic behavior if they were not properly modeled such as system events and events coming from motion sensors (e.g., accelerometer, gyroscope, and magnetometer). DroidbotX will be extended in the future to include more system events.

6.5 List of Scholarly Publications

The list of publications related to the research undertaken in this thesis is as follows:

Yasin, H.N.; Hamid, S.H.A.; Yusof, R.J.R.; Hamzah, M. An Empirical Analysis of Test Input Generation Tools for Android Apps through a Sequence of Events. Symmetry 2020, 12,

Yasin, H.N.; Hamid, S.H.A.; Raja Yusof, R.J. DroidbotX: Test Case Generation Tool for Android Applications Using Q-Learning. Symmetry 2021, 13, 310.

Yasin, H. N., Hamid S. H., & Raja-Yusof, R. J., (2021) GUI Testing Frameworks for Android apps: Taxonomy and Open Research Issues. (draft)

REFERENCES

Adamo, D., Khan, M. K., Koppula, S., & Bryce, R. (2018). Reinforcement learning for Android GUI testing. Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation,

Adamo, D., Nurmuradov, D., Piparia, S., & Bryce, R. (2018). Combinatorial-based event sequence testing of Android applications. *Information and Software Technology*, *99*, 98-117.

Adamsen, C. Q., Mezzetti, G., & Møller, A. (2015). Systematic execution of android test suites in adverse conditions. Proceedings of the 2015 International Symposium on Software Testing and Analysis,

Amalfitano, D., Amatucci, N., Fasolino, A. R., & Tramontana, P. (2015a). A conceptual framework for the comparison of fully automated gui testing techniques. 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW),

Amalfitano, D., Amatucci, N., Fasolino, A. R., & Tramontana, P. (2015b). AGRippin: a novel search based testing technique for Android applications. Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile,

Amalfitano, D., Fasolino, A. R., & Tramontana, P. (2011). A gui crawling-based technique for android mobile application testing. 2011 IEEE fourth international conference on software testing, verification and validation workshops,

Amalfitano, D., Fasolino, A. R., Tramontana, P., & Amatucci, N. (2013). Considering context events in event-based testing of mobile applications. 2013 IEEE sixth international conference on software testing, verification and validation workshops,

Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Imparato, G. (2012). A toolset for GUI testing of Android applications. 2012 28th IEEE International Conference on Software Maintenance (ICSM),

Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Memon, A. M. (2012). Using GUI ripping for automated testing of Android applications. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering,

Amalfitano, D., Fasolino, A. R., Tramontana, P., & Robbins, B. (2013). Testing android mobile applications: Challenges, strategies, and approaches. In *Advances in Computers* (Vol. 89, pp. 1-52). Elsevier.

Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., & Memon, A. M. (2014). MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software*, *32*(5), 53-59.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., Mcminn, P., & Bertolino, A. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, *86*(8), 1978-2001.

Anand, S., Naik, M., Harrold, M. J., & Yang, H. (2012). Automated concolic testing of smartphone apps. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering,

AppBrain. (2009). *Monetize, advertise and analyze Android apps | AppBrain*. TheAppBrain. Retrieved 20 December 2019 from https://www.appbrain.com/

Arcuri, A. (2011). A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Transactions on Software Engineering*, *38*(3), 497-519.

Arel, I., Liu, C., Urbanik, T., & Kohls, A. G. (2010). Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems*, *4*(2), 128-135.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26-38.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. Machine learning, 47(2), 235-256.

Azim, T., & Neamtiu, I. (2013). 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; pp. 641–660.

Baek, Y.-M., & Bae, D.-H. (2016). Automated model-based Android GUI testing using multi-level GUI comparison criteria. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering,

Banerjee, I., Nguyen, B., Garousi, V., & Memon, A. (2013). Graphical user interface (GUI) testing: Systematic mapping and repository. *Information and Software Technology*, *55*(10), 1679-1694.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2014). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, *41*(5), 507-525.

Bauersfeld, S., & Vos, T. E. (2014). User interface level testing with TESTAR; what about more sophisticated action specification and selection? SATToSE,

Borges, N. P., & Zeller, A. (2019). Why Does this App Need this Data? Automatic Tightening of Resource Access. 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), Bu, X., Rao, J., & Xu, C.-Z. (2009). A reinforcement learning approach to online web systems auto-configuration. 2009 29th IEEE International Conference on Distributed Computing Systems,

Canfora, G., Di Sorbo, A., Mercaldo, F., & Visaggio, C. A. (2016). Exploring mobile user experience through code quality metrics. International Conference on Product-Focused Software Process Improvement,

Chaffey, D. (2018). *Mobile marketing statistics compilation | Smart Insights*. S. Insights. Retrieved 20 December 2019, from https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

Chen, T. Y., Kuo, F.-C., Merkel, R. G., & Tse, T. (2010). Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, *83*(1), 60-66.

Choi, W., Necula, G., & Sen, K. (2013). Guided gui testing of android apps with minimal restart and approximate learning. Acm Sigplan Notices,

Choudhary, S. R., Gorla, A., & Orso, A. (2015). Automated test input generation for android: Are we there yet?(e). 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE),

Chui, M., Manyika, J., Miremadi, M., Henke, N., Chung, R., Nel, P., & Malhotra, S. (2018). Notes from the AI frontier: Applications and value of deep learning. *McKinsey global institute discussion paper, April*.

Clapp, L., Bastani, O., Anand, S., & Aiken, A. (2016). Minimizing GUI event traces. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering,

Dashevskyi, S., Gadyatskaya, O., Pilgun, A., & Zhauniarovich, Y. (2018). The influence of code coverage metrics on automated testing efficiency in android. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security,

Eclipse. (2010). *Eclipse MoDisco | The Eclipse Foundation*. Eclipse Foundation. Retrieved 20 December 2019, from https://www.eclipse.org/MoDisco/

Esparcia-Alcázar, A. I., Almenar, F., Martínez, M., Rueda, U., & Vos, T. (2016). Q-learning strategies for action selection in the TESTAR automated testing tool. *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*, 130-137.

F-Droid. (2010). *F-Droid - Free and Open Source Android App Repository*. Retrieved 10 December 2019, from https://f-droid.org/

Fazzini, M., Freitas, E. N. D. A., Choudhary, S. R., & Orso, A. (2017). Barista: A technique for recording, encoding, and running platform independent android tests. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST),

Fewster, M., & Graham, D. (1999). *Software test automation*. Addison-Wesley Reading.

Freke, J. (2013). Smali, an assembler/disassembler for Android's dex format. Retrieved 26 December 2020, from https://github.com/JesusFreke/smali.

Gomez, L., Neamtiu, I., Azim, T., & Millstein, T. (2013). Reran: Timing-and touch-sensitive record and replay for android. Proceedings of the 2013 International Conference on Software Engineering,

Google. (2019a). *Activities | Android Developers*. Google Inc. Retrieved 16 December 2019, from https://developer.android.com/guide/components/activities.html

Google. (2019b). *Application Fundamentals | Android Developers*. Google Inc. Retrieved 20 December 2019, from https://developer.android.com/guide/components/fundamentals?hl=en

Google. (2019c). *Command line tools | Android Developers*. Google Inc. Retrieved 16 December 2019, from https://developer.android.com/studio/command-line

Google. (2019d). *Crashes | Android Developers*. Google. Retrieved 25 December 2019, from https://developer.android.com/topic/performance/vitals/crash

Google. (2019e). *Espresso | Android Developers*. Google. Retrieved 10 December 2019, from https://developer.android.com/training/testing/espresso

Google. (2019f). *Introduction to Activities | Android Developers*. Google. Retrieved 25 2019, December from https://developer.android.com/guide/components/activities/intro-activities?hl=en

Google. (2019g). *monkeyrunner | Android Developers*. Google. Retrieved 10 December 2019, from https://developer.android.com/studio/test/monkeyrunner

Google. (2019h). *Platform Architecture | Android Developers*. Retrieved 22 December 2019, from https://developer.android.com/guide/platform/

Google. (2019i). *UI Automator | Android Developers*. Google. Retrieved 10 December 2019, from https://developer.android.com/training/testing/ui-automator

Google. (2019j). *UI/Application Exerciser Monkey | Android Developers*. Google. Retrieved 10 December 2019, from https://developer.android.com/studio/test/monkey

Google. (2019k). *Understand the Activity Lifecycle | Android Developers*. Google. Retrieved 25 December 2019, from https://developer.android.com/guide/components/activities/activity-lifecycle.html

Gu, T., Cao, C., Liu, T., Sun, C., Deng, J., Ma, X., & Lü, J. (2017). Aimdroid: Activity-insulated multi-level automated testing for android applications. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME),

Gu, T., Sun, C., Ma, X., Cao, C., Xu, C., Yao, Y., Zhang, Q., Lu, J., & Su, Z. (2019). Practical GUI testing of Android applications via model abstraction and refinement. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE),

Gunasekaran, S., & Bargavi, V. (2015). Survey on automation testing tools for mobile applications. *International Journal of Advanced Engineering Research and Science*, *2*(11), 2349-6495.

Hao, S., Liu, B., Nath, S., Halfond, W. G., & Govindan, R. (2014). PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. Proceedings of the 12th annual international conference on Mobile systems, applications, and services,

Haoyin, L. (2017). Automatic android application GUI testing—A random walk approach. 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET),

Harman, M., Mansouri, S. A., & Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, *45*(1), 11.

Hu, C., & Neamtiu, I. (2011). Automating GUI testing for Android applications. Proceedings of the 6th International Workshop on Automation of Software Test,

Hu, G., Yuan, X., Tang, Y., & Yang, J. (2014). Efficiently, effectively detecting mobile app bugs with appdoctor. Proceedings of the Ninth European Conference on Computer Systems,

Hu, Z., Ma, Y., & Huang, Y. (2017). DroidWalker: Generating Reproducible Test Cases via Automatic Exploration of Android Apps. *arXiv preprint arXiv:1710.08562*.

Huang, C.-Y., Chiu, C.-H., Lin, C.-H., & Tzeng, H.-W. (2015). Code coverage measurement for Android dynamic analysis tools. 2015 IEEE International Conference on Mobile Services,

IDC. (2019). *IDC - Smartphone Market Share - OS*. IDC. Retrieved 16 December 2019, from https://www.idc.com/promo/smartphone-market-share

Imparato, G. (2015). A combined technique of GUI ripping and input perturbation testing for Android apps. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,

Jensen, C. S., Prasad, M. R., & Møller, A. (2013). Automated testing with targeted event sequence generation. Proceedings of the 2013 International Symposium on Software Testing and Analysis,

Joorabchi, M. E., Mesbah, A., & Kruchten, P. (2013). Real challenges in mobile app development. 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,

Kaasila, J., Ferreira, D., Kostakos, V., & Ojala, T. (2012). Testdroid: automated remote UI testing on Android. Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, *4*, 237-285.

Kamiura, M., & Sano, K. (2017). Optimism in the face of uncertainty supported by a statistically-designed multi-armed bandit algorithm. *Biosystems*, *160*, 25-32.

Kayes, A., Kalaria, R., Sarker, I. H., Islam, M., Watters, P. A., Ng, A., Hammoudeh, M., Badsha, S., & Kumara, I. (2020). A Survey of Context-Aware Access Control Mechanisms for Cloud and Fog Networks: Taxonomy and Open Research Issues. *Sensors*, *20*(9), 2464.

Khalid, H., Shihab, E., Nagappan, M., & Hassan, A. E. (2014). What do mobile app users complain about? *IEEE software*, *32*(3), 70-77.

Kim, J., Kwon, M., & Yoo, S. (2018). Generating test input with deep reinforcement learning. 2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST),

King, J. C. (1975). A new approach to program testing. *Acm Sigplan Notices*, *10*(6), 228-233.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, *28*(8), 721-734.

Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., & Lo, D. (2015). Understanding the test automation culture of app developers. 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST),

Koenig, S., & Simmons, R. G. (1993). Complexity analysis of real-time reinforcement learning. AAAI,

Koroglu, Y., Sen, A., Muslu, O., Mete, Y., Ulker, C., Tanriverdi, T., & Donmez, Y. (2018). QBE: QLearning-based exploration of android applications. 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST),

Kowalczyk, E., Cohen, M. B., & Memon, A. M. (2018). Configurations in Android testing: they matter. Proceedings of the 1st International Workshop on Advances in Mobile App Analysis,

Kropp, M., & Morales, P. (2010). Automated GUI testing on the Android platform. *on Testing Software and Systems: Short Papers*, 67.

Li, K., & Wu, M. (2004). *Effective GUI test automation: developing an automated GUI testing tool.*

Li, L., Bissyandé, T. F., Klein, J., & Le Traon, Y. (2016). An investigation into the use of common libraries in android apps. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Li, Y., Yang, Z., Guo, Y., & Chen, X. (2017). DroidBot: a lightweight UI-guided test input generator for Android. 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C),

Li, Y., Yang, Z., Guo, Y., & Chen, X. (2019). A Deep Learning based Approach to Automated Android App Testing. *arXiv preprint arXiv:1901.02633*.

Lin, Y.-D., Chu, E. T.-H., Yu, S.-C., & Lai, Y.-C. (2013). Improving the accuracy of automated GUI testing for embedded systems. *IEEE software*, *31*(1), 39-45.

Lin, Y.-D., Rojas, J. F., Chu, E. T.-H., & Lai, Y.-C. (2014). On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering*, *40*(10), 957-970.

Linares-Vasquez, M., Vendome, C., Luo, Q., & Poshyvanyk, D. (2015). How developers detect and fix performance bottlenecks in Android apps. 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME),

Linares-Vásquez, M. (2015). Enabling testing of android apps. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,

Linares-Vásquez, M., Moran, K., & Poshyvanyk, D. (2017). Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME),

Linares-Vásquez, M., White, M., Bernal-Cárdenas, C., Moran, K., & Poshyvanyk, D. (2015). Mining android app usages for generating actionable gui-based execution scenarios. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories,

Liu, C. H., Lu, C. Y., Cheng, S. J., Chang, K. Y., Hsiao, Y. C., & Chu, W. M. (2014). Capture-replay testing for android applications. 2014 International Symposium on Computer, Consumer and Control,

Liu, Z., Gao, X., & Long, X. (2010). Adaptive random testing of mobile application. 2010 2nd International Conference on Computer Engineering and Technology,

Lonza, A. (2019). *Reinforcement learning algorithms with Python: learn, understand, and develop smart algorithms for addressing AI challenges*. Packt Publishing.

Machiry, A., Tahiliani, R., & Naik, M. (2013). Dynodroid: An input generation system for android apps. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering,

Mahmood, R., Mirzaei, N., & Malek, S. (2014). Evodroid: Segmented evolutionary testing of android apps. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering,

Maji, A. K., Hao, K., Sultana, S., & Bagchi, S. (2010). Characterizing failures in mobile oses: A case study with android and symbian. 2010 IEEE 21st International Symposium on Software Reliability Engineering, Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. Proceedings of the 15th ACM Workshop on Hot Topics in Networks,

Mao, K., Harman, M., & Jia, Y. (2016). Sapienz: Multi-objective automated testing for Android applications. Proceedings of the 25th International Symposium on Software Testing and Analysis,

Mariani, L., Pezze, M., Riganelli, O., & Santoro, M. (2012). Autoblacktest: Automatic black-box testing of interactive applications. 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation,

Mariani, L., Pezzè, M., Riganelli, O., & Santoro, M. (2011). AutoBlackTest: a tool for automatic black-box testing. 2011 33rd International Conference on Software Engineering (ICSE),

Martin, W., Sarro, F., & Harman, M. (2016). Causal impact analysis for app releases in google play. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering,

Memon, A. (2001). *Comprehensive Framework for Testing Graphical User Interfaces*. University of Pittsburgh Pittsburgh.

Memon, A. (2002). GUI testing: Pitfalls and process. *Computer*(8), 87-88.

Memon, A. (2019). *Advances in Computers* (Vol. 112). Academic Press.

Memon, A., Soffa, M. L., & Pollack, M. (2001). Coverage criteria for GUI testing. *ACM SIGSOFT Software Engineering Notes*, *26*(5), 256-267.

Memon, A. M. (2003). Advances in GUI testing. *Advances in Computers*, *58*, 149-201.

Mirzaei, N., Bagheri, H., Mahmood, R., & Malek, S. (2015). Sig-droid: Automated system input generation for android applications. 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE),

Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., & Malek, S. (2016). Reducing combinatorics in GUI testing of android applications. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE),

Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., & Poshyvanyk, D. (2016). Automatically discovering, reporting and reproducing android application crashes. 2016 IEEE international conference on software testing, verification and validation (icst),

Morgado, I. C., & Paiva, A. C. (2015). Testing approach for mobile applications through reverse engineering of UI patterns. 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW),

Morrison, G. C., Inggs, C. P., & Visser, W. (2012). Automated coverage calculation and test case generation. Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, Muccini, H., Di Francesco, A., & Esposito, P. (2012). Software testing of mobile applications: Challenges and future research directions. Proceedings of the 7th International Workshop on Automation of Software Test,

Musa, J. (1987). Software Quality and Reliabity Basics. *AT&T Bell Laboratories*, *1*(1), 114-115.

Nolan, G. (2015). *Agile Android*. Apress.

Packard, H. (2015). *Failing to meet mobile app user expectations: a mobile user survey* (Tech. rep., Issue.

Palomba, F., Linares-Vasquez, M., Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D., & De Lucia, A. (2015). User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. 2015 IEEE international conference on software maintenance and evolution (ICSME),

Perry, D. E., Sim, S. E., & Easterbrook, S. M. (2004). Case studies for software engineers. Proceedings. 26th International Conference on Software Engineering,

Pilgun, A., Gadyatskaya, O., Zhauniarovich, Y., Dashevskyi, S., Kushniarou, A., & Mauw, S. (2020). Fine-grained code coverage measurement in automated black-box Android testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *29*(4), 1-35.

Ravindranath, L., Nath, S., Padhye, J., & Balakrishnan, H. (2014). Automatic and scalable fault detection for mobile applications. Proceedings of the 12th annual international conference on Mobile systems, applications, and services,

Reda, R., & Josefson, H. (2014). *RobotiumTech/robotium: Android UI Testing*. RobotiumTech. Retrieved 10 December 2019, from https://github.com/RobotiumTech/robotium

Rubinov, K., & Baresi, L. (2018). What Are We Missing When Testing Our Android Apps? *Computer*, *51*(4), 60-68.

Sadeghi, A., Jabbarvand, R., & Malek, S. (2017). Patdroid: permission-aware gui testing of android. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,

Saeed, A., Ab Hamid, S. H., & Sani, A. A. (2017). Cost and effectiveness of search-based techniques for model-based testing: an empirical analysis. *International Journal of Software Engineering and Knowledge Engineering*, *27*(04), 601-622.

Salihu, I.-A., Ibrahim, R., Ahmed, B. S., Zamli, K. Z., & Usman, A. (2019). AMOGA: a static-dynamic model generation strategy for mobile apps testing. *IEEE Access*, *7*, 17158-17173.

Sauce. (2013). *Appium | Automation for iOS, Android, and Windows Apps*. Retrieved 16 December 2019, from http://appium.io/

Sen, K. (2007). Concolic testing. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering,

Septian, I., & Alianto, R. S. (2018). Comparison Analysis of Android GUI Testing Frameworks by Using an Experimental Study. *Procedia Computer Science*, *135*, 736-748.

Shahamiri, S. R., Kadir, W. M. N. W., & Mohd-Hashim, S. Z. (2009). A comparative study on automated software test oracle methods. 2009 Fourth International Conference on Software Engineering Advances,

Sharma, C., Sabharwal, S., & Sibal, R. (2014). A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*.

Shebaro, B., Oluwatimi, O., & Bertino, E. (2014). Context-based access control systems for mobile devices. *IEEE Transactions on Dependable and Secure Computing*, *12*(2), 150-163.

Singh, S., Gadgil, R., & Chudgor, A. (2014). Automated Testing of mobile applications using scripting Technique: A study on Appium. *International Journal of Current Engineering and Technology (IJCET)*, *4*(5), 3627-3630.

Song, W., Qian, X., & Huang, J. (2017). EHBDroid: Beyond GUI testing for Android applications. Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering,

Statista. (2019). *App stores: number of apps in leading app stores 2019 | Statista*. Retrieved 14 December 2019, from https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

Su, T. (2016). FSMdroid: guided GUI testing of android apps. 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C),

Su, T., Fan, L., Chen, S., Liu, Y., Xu, L., Pu, G., & Su, Z. (2020). Why my app crashes understanding and benchmarking framework-specific exceptions of android apps. *IEEE Transactions on Software Engineering*. on, pp. 557-568, 2020

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., & Su, Z. (2017). Guided, stochastic model-based GUI testing of Android apps. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,

Sutton, R. S., & Barto, A. G. (1998). *Introduction to reinforcement learning* (Vol. 135). MIT press Cambridge.

Takala, T., Katara, M., & Harty, J. (2011). Experiences of system-level model-based GUI testing of an Android application. 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation,

Taylor-Sakyi, K. (2016). Reliability Testing Strategy-Reliability in Software Engineering. *arXiv preprint arXiv:1605.01097*.

TheAppBrain. (2019). *Number of Android applications on the Google Play store | AppBrain*. TheAppBrain. Retrieved 16 December 2019 from https://www.appbrain.com/stats/number-of-android-apps

Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, *22*(5), 297-312.

Vuong, T. A. T., & Takada, S. (2018). A reinforcement learning based approach to automated testing of Android applications. Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation,

Wang, P., Liang, B., You, W., Li, J., & Shi, W. (2014). Automatic Android GUI traversal with high coverage. 2014 Fourth International Conference on Communication Systems and Network Technologies,

Wang, W., Li, D., Yang, W., Cao, Y., Zhang, Z., Deng, Y., & Xie, T. (2018). An empirical study of android test generation tools in industrial cases. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering,

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279-292.

Wen, H.-L., Lin, C.-H., Hsieh, T.-H., & Yang, C.-Z. (2015). Pats: A parallel gui testing framework for android applications. 2015 IEEE 39Th annual computer software and applications conference,

Xie, Q., & Memon, A. M. (2006). Studying the characteristics of a" Good" GUI test suite. 2006 17th International Symposium on Software Reliability Engineering,

Yang, S., Huang, S., & Hui, Z. (2019). Theoretical Analysis and Empirical Evaluation of Coverage Indictors for Closed Source APP Testing. *IEEE Access*, *7*, 162323-162332.

Yang, W., Prasad, M. R., & Xie, T. (2013). A grey-box approach for automated GUI-model generation of mobile applications. International Conference on Fundamental Approaches to Software Engineering,

Yu, S., & Takada, S. (2016). Mobile application test case generation focusing on external events. Proceedings of the 1st International Workshop on Mobile Development,

Zaeem, R. N., Prasad, M. R., & Khurshid, S. (2014). Automated generation of oracles for testing user-interaction features of mobile apps. 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation,

Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., & Xie, T. (2016). Automated test input generation for android: Are we really there yet in an industrial case? Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering,

Zhauniarovich, Y., Philippov, A., Gadyatskaya, O., Crispo, B., & Massacci, F. (2015). Towards black box testing of android apps. 2015 10th International Conference on Availability, Reliability and Security, Zheng, H., Li, D., Liang, B., Zeng, X., Zheng, W., Deng, Y., Lam, W., Yang, W., & Xie, T. (2017). Automated test input generation for android: Towards getting there in an industrial case. 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP),

Zhou, Z., Li, X., & Zare, R. N. (2017). Optimizing chemical reactions with deep reinforcement learning. *ACS central science*, *3*(12), 1337-1344.

Zhu, H., Ye, X., Zhang, X., & Shen, K. (2015). A context-aware approach for dynamic gui testing of android applications. 2015 IEEE 39th Annual Computer Software and Applications Conference,

Zitzler, E., & Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE transactions on Evolutionary Computation*, *3*(4), 257-271.