Testing mobile applications via automated exploration of graphical user interface and textual user input generation

Arnatovich, Yauhen Leanidavich

2019

Arnatovich, Y. L. (2019). Testing mobile applications via automated exploration of graphical user interface and textual user input generation. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/89265>

<https://doi.org/10.32657/10220/48041>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

**TESTING MOBILE APPLICATIONS VIA AUTOMATED
EXPLORATION OF GRAPHICAL USER INTERFACE
AND TEXTUAL USER INPUT GENERATION**

YAUHEN LEANIDAVICH ARNATOVICH

School of Electrical and Electronic Engineering

2019

**TESTING MOBILE APPLICATIONS VIA
AUTOMATED EXPLORATION OF GRAPHICAL
USER INTERFACE AND TEXTUAL USER INPUT
GENERATION**

YAUHEN LEANIDAVICH ARNATOVICH

School of Electrical and Electronic Engineering

The thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

15 April 2019

.....

Date

.....

Yauhen Leanidavich Arnatovich

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

15 April 2019

.....
Date

.....
Wang Lipo

Authorship Attribution Statement

This thesis contains material from two papers published in the following peer-reviewed journals where I am the first author.

Chapter 5 is published as **Y. L. Arnatovich**, L. Wang, N. M. Ngo and C. Soh, “Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customised input generation,” in *Software: Practice and Experience*, vol. 48, pp. 1107–1142, 2018. DOI: <https://doi.org/10.1002/spe.2564>

The contributions of the co-authors are as follows:

- A/Prof Lipo Wang provided the initial direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by Dr. Ngoc Minh Ngo.
- I co-designed the study with A/Prof Lipo Wang and conducted the experiment.
- Mr Charlie Soh assisted in the collection of the experimental data.

Appendix A is published as **Y. L. Arnatovich**, L. Wang, N. M. Ngo and C. Soh, “A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation,” in *IEEE Access*, vol. 6, pp. 12382-12394, 2018. DOI: 10.1109/ACCESS.2018.2808340

The contributions of the co-authors are as follows:

- A/Prof Lipo Wang suggested the materials area and edited the manuscript drafts.
- I wrote the drafts of the manuscript. The manuscript was revised together with Dr. Ngoc Minh Ngo.
- I performed all the materials synthesis, and conducted evaluation of the results.
- Mr Charlie Soh collected the experimental data.

15 April 2019

.....
Date

.....
Yauhen Leanidavich Arnatovich

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Dr. **Wang Lipo** and Dr. **Tan Hee Beng Kuan**, for their continuous help, support, and thorough guidance during my PhD coursework and research.

I also thank to my friends and PhD colleagues, **Charlie Soh Zhan Yi**, Dr. **Cristal Ngo Minh Ngoc**, Dr. **Lwin Khin Shar**, and Dr. **Sun Ding**, for their kind help and useful advices in contributing to the research by doing an experimental work, and writing of the papers.

I wish to thank my wife, Dr. **Liu Aifei**, for her love, support, understanding, and encouragement during my PhD journey. I am deeply indebted to my parents, sister and her family for their unconditional love, financial and emotional support.

Table of Contents

Statement of Originality	i
Supervisor Declaration Statement	ii
Authorship Attribution Statement	iii
Acknowledgments	iv
Table of Contents	v
List of Abbreviations	ix
Summary	x
List of Figures	xii
List of Tables	xiii
1 Introduction	1
1.1 Motivations	1
1.2 Objectives	3
1.3 Major Contributions of Thesis	3
1.4 Organization of Thesis	4
2 Background	6
2.1 Overview of GUI: Structure, Components and Elements	6
2.2 Nature of GUI Testing: Approaches, Techniques and Methods	9
3 Literature Review	17
3.1 Related Work	17
3.1.1 Autodroid	17
3.1.2 AndroFrame	18
3.1.3 Mobolic	20
3.1.4 Stoa	21

3.1.5	AimDroid	23
3.1.6	EHBDroid	24
3.1.7	PATDroid	25
3.1.8	Xdroid	26
3.1.9	LAND	27
3.1.10	DroidWalker	28
3.1.11	Sapienz	29
3.1.12	GUICC	30
3.1.13	DroidDEV	32
3.1.14	MCrawlT	33
3.1.15	GAT	34
3.1.16	MobiGUITAR	35
3.1.17	SIG-Droid	37
3.1.18	EvoDroid	38
3.1.19	ADAutomation	39
3.1.20	SwiftHand	40
3.1.21	Collider	41
3.1.22	A ³ E	42
3.1.23	ORBIT	44
3.1.24	Dynodroid	45
3.1.25	ACTEve	46
3.2	Discussion	47
3.2.1	Effectiveness	47
3.2.2	Efficiency	48
3.3	Conclusion	50
4	Automated Functional Testing of Mobile Apps GUI	51
4.1	Summary	51
4.2	Introduction	51
4.3	Proposed Approach: DroidDEV	53
4.3.1	Problem statement	53
4.3.2	Definitions	54
4.3.3	Novel aspects	55
4.3.4	Overview	57
4.3.5	Components	58
4.4	Design and Implementation of DroidDEV: Practical Experience	63
4.4.1	What is the f-UFG model?	64
4.4.2	How do we build the f-UFG model?	64
4.4.3	How do we represent the f-UFG model?	64

4.4.4	How do we dump and exercise the UI widgets?	65
4.4.5	How do we implement the GUI traversal technique using A*? .	65
4.4.6	How do we generate the UI-context-aware textual user inputs? .	66
4.5	DroidDEV: Demo	68
4.6	Empirical Evaluation	69
4.6.1	Experimental setup	70
4.6.2	Code coverage analysis with EMMA	71
4.6.3	Experimental results	73
4.7	Discussion	76
4.8	Conclusion	77

5 Automated Textual Input Generation for Functional Testing of Mobile

Apps GUI		78
5.1	Summary	78
5.2	Introduction	79
5.3	Proposed Approach: Mobolic	80
5.3.1	Problem statement	81
5.3.2	Definitions	81
5.3.3	Novel aspects	83
5.3.4	Overview	83
5.3.5	Components	88
5.4	Design and Implementation of Mobolic: Practical Experience	95
5.4.1	How do we relate editable UI widgets with the Java code? . . .	95
5.4.2	How do we implement concolic execution using MATLAB symbolic engine?	96
5.4.3	How do we generate concrete textual user inputs?	101
5.5	Mobolic: Demo	104
5.6	Empirical Evaluation	107
5.6.1	Experimental dataset	108
5.6.2	Experimental environment	109
5.6.3	Experiment design	110
5.6.4	Experiment operation	112
5.6.5	Experimental results	113
5.6.6	Comparison of Mobolic with other mobile GUI-testing tools . .	118
5.7	Discussion	119
5.8	Threats to Validity	121
5.9	Related Work	123
5.9.1	Random testing	128
5.9.2	Model-based testing	129

5.9.3	Dynamic symbolic execution	131
5.10	Conclusion	132
6	Conclusion and Recommendations	133
6.1	Conclusion	133
6.2	Recommendations for Further Research	134
	Author's Publications	136
	Bibliography	137
	Appendix A	161

List of Abbreviations

<i>GUI</i>	graphical user interface
<i>UFG</i>	GUI-flow graph
<i>f-UFG</i>	finite-state UFG without loops
<i>UI</i>	hierarchy view of widgets/controls which define a screen layout
<i>app UI</i>	foreground (visible) UI on device screen
<i>AUT</i>	app/application under test
<i>Monkey</i>	dumb Monkey (default Android Monkey)
<i>UID statement</i>	user-input dependent statement
<i>symbolic PC</i>	symbolic path constraint
<i>CDG</i>	Control Dependence graph
<i>r-CDG</i>	reduced Control Dependence graph
<i>DDG</i>	Data Dependence graph

Summary

A Graphical User Interface (GUI) is acknowledged as a crucial component of an event-driven software (e.g., mobile apps). In the event-driven software, the GUI usually contains hundreds or even thousands of elements. As such, a large part of app functionality, and thus its code, is usually dedicated to a user interface, so its testing becomes an essential part of the software development lifecycle significantly improving quality of software. During a testing phase, the GUI can be tested by executing each event individually and observing its behaviour. However, it is not a trivial task since the behaviour of an event handler may depend on a GUI internal state, state of other entities (objects, event handlers) and an external environment. Furthermore, an outcome of the event handler execution may vary depending on a particular sequence of preceding events. As a result, each GUI event needs to be tested in a context of different states via generating and executing various sequences of the GUI events.

Modern mobile apps have a highly interactive nature and complex GUI structure. As such, an automated GUI testing of mobile apps is a daunting task for developers and testers. Often the GUI testing is done manually where all possible combinations of the GUI elements for a given app screen are manually tested for functional correctness and aesthetic quality. The manual GUI-testing is no doubt an effective approach, however, it is inefficient, i.e., time-consuming, error-prone, and usually not complete, especially for a large software with complex GUIs. So, to facilitate manual testing, various automated testing approaches have been introduced such as model-based testing, concolic testing, search-based testing, evolutionary testing, and combinatorial testing. However, for automated testing, there is a challenge to generate tests with high coverage, as well as to maintain a reasonable execution time. These are hindered by the non-trivial structure, and highly interactive nature of mobile apps GUIs so that an achieved code coverage is generally low while execution time is high.

To address these problems, in this thesis, we propose a novel automated GUI testing technique for mobile apps. Our technique automatically generates tests with high coverage within a reasonable time by combining the online testing with active learning, and customed¹ textual input generation. By employing the online testing with active learning, our technique systematically explores an app GUI without falling in a loop by generating relevant events “on-the-fly” which are followed by immediate execution. By employing the customed input generation, our technique automatically generates relevant user inputs such as UI-context-aware, user-predefined, concrete, or random ones to explore constrained app screens. We use various real-world open-source Android apps for evaluation. Our experimental results show higher effectiveness and efficiency of our technique in comparison with the other state-of-the-art and practice automated testing techniques regarding an achieved code coverage and execution time.

¹“customed” is an abbreviation which is constructed from the letters of the following words: “concrete”, “user-specific”, and “automated”; its name implies that the “customed” input generation mechanism can *automatically* generate non-trivial *concrete* and *user-specific* inputs.

List of Figures

2.1	A view hierarchy which defines an Android app interface structure (layout)	7
2.2	Example of standard Android app UI elements	8
2.3	Model-based testing: Overview of approach	13
2.4	Finite state machine: Example of system states and transitions	14
2.5	Concolic execution: Mix of concrete and symbolic (abstract) execution .	15
4.1	Example UFG of mobile app	55
5.1	App code example with UID statements	82
5.2	r-CDG example of app code	82
5.3	Overview of Mobolic workflow	84
5.4	Architectural design of customated input generation mechanism	97
5.5	Example mobile app UI	97
5.6	Example code of mobile app with UID conditional statement	98
5.7	Example code of mobile app with instrumented UID conditional statement	99
5.8	Example MATLAB code for solving UID conditional statement	100
5.9	Example UFG of mobile app	104
6.1	Dalvik virtual machine: Creation and usage	165
6.2	Difference in process of transformation Java source code (.java) into Java bytecode (.class), and Dalvik executable (.dex)	166
6.3	Difference in internal structure of Java bytecode (.class), and Dalvik executable (.dex)	167
6.4	Monkey tests results before log-transformation: Log-normal distribution of data values	172
6.5	Monkey tests results after log-transformation: Approximately normally distributed data values	173
6.6	Number of apps failing Monkey tests for 26 categories tested	179

List of Tables

4.1	Types of events supported by DroidDEV	58
4.2	Code coverage for DroidDEV, ManualTest, MobiGUITAR, Monkey, and time taken by DroidDEV	75
5.1	Types of events supported by Mobolic SAT	90
5.2	Summary of characteristics of 10 apps tested	108
5.3	Android emulator device specifications	110
5.4	Code coverage for Mobolic, Human, Sapienz, ACTEve, MobiGUITAR, and time taken by Mobolic	114
5.5	Code coverage, and execution time for Mobolic and ACTEve	118
5.6	Code coverage for Mobolic, DroidDEV, Human (ManualTest), Monkey, Sapienz, ACTEve, and MobiGUITAR and time taken for Mobolic, and DroidDEV	120
5.7	Comparison of specifications of mobile testing tools	124
5.8	Comparison of characteristics of testing tools	126
5.9	Comparison of GUI exploration strategies of testing tools	127
6.1	Android emulator device specifications	168
6.2	Number of apps failing Monkey test for 26 categories tested	177
6.3	Statistic of original Google Chrome app for five replications (Monkey tests)	178
6.4	One-sample t-test results for original Google Chrome app	178
6.5	Two-sample t-test results for Apktool, Dex2jar, and Soot	181

Chapter 1

Introduction

1.1 Motivations

Nowadays, there is a notable shift to the favour of mobile personal devices in numerous traditional application areas such as e-mail, social networking, entertainment, and e-commerce. This new wave has prompted an explosive growth in the number and popularity of various mobile applications (or simply “apps”). However, due to pressing time to market and, usually, insufficient human resources, mobile apps are developed without adequate testing of their desired functionality. As such, there is an increasing demand for automated testing of mobile apps.

A central challenge in automated testing is a generation of tests with high coverage. Due to the specific non-trivial structure and highly interactive nature of the mobile apps graphical user interfaces (GUIs) [1–3], existing automated GUI testing techniques [4–7] for desktop applications are not suitable “*as are*” for the GUI testing of mobile apps. As such, in recent years, various automated GUI testing techniques for mobile apps have been developed [2, 8–14]. However, practice shows that they are still too immature to generate tests with high coverage [14, 15] in an automated manner. This problem is mainly because they do not simultaneously address both of the following major problems. The first problem, **Problem 1**, relates to the performance of the testing technique, i.e., efficiency and effectiveness of an implemented GUI exploration strategy. The second problem, **Problem 2**, relates to the generation of the relevant textual user inputs, i.e., adequate to the app state textual user input, e.g., specific numeric or textual value. Next, we describe the problems in detail.

Problem 1 In general, mobile apps have a non-trivial GUI structure, i.e., their UI event-flow models are not finite state and/or containing loops. Therefore, it is a non-trivial task to perform automated testing of the mobile apps GUIs. For example, existing tools for GUI testing of mobile apps build GUI models and use them “*as are*” for apps GUI

testing. However, due to the non-trivial GUI structure of the mobile apps, their models commonly have an infinite number of UIs (visual screens), or may have loops for all but trivial apps. As such, an automated testing process may infinitely exercise an app GUI, unless a termination condition is explicitly specified by a user [16]. So, to mitigate this problem, the existing mobile testing techniques [15, 16] manually specify the termination condition for their testing procedure such as exact execution time, the predefined number of events to be injected, or depth of exploration in a UI event-flow model. However, using such termination conditions, the existing techniques may overlook specific app UIs leaving their functionality untested, mainly if the app UIs are located at deep levels in the UI event-flow model.

In nature, a methodology of our approach is generic and can be realised for any software with GUI including mobile and PC. That is, our proposed finite state GUI-model without loops (*f-UFG*), and textual input generation (*customated*) can be implemented for different software which has GUI. However, implementation of the approach will differ depending on a platform (i.e., mobile [e.g., Android, iOS] or PC [e.g., Windows, Unix-based, macOS]) where testing is being performed. For example, to build the *f-UFG* model, mobile and PC software will require different GUI rippers for the GUI elements properties extraction, and also the *customated* input generation will require appropriate tools for an app code instrumentation, static code analysis, and dynamic symbolic execution.

Problem 2 In fact, mobile apps' GUIs are highly interactive. Therefore, mobile apps commonly require sensible textual input from users. As such, to adequately explore an app GUI in an automated manner, non-trivial relevant textual user inputs must be generated. To the best of our knowledge, this remains a challenging task. For example, many existing mobile GUI testing techniques [15] either generate inputs randomly and/or provide them manually during testing. However, random inputs are too trivial, while manual efforts are time-consuming and error-prone. Therefore, to automatically generate relevant user inputs, the existing techniques [8, 17, 18] use symbolic execution. However, these techniques tend to symbolically execute all the program paths existing in an app, which is impractical for all but trivial apps since symbolic execution has inherent limitations such as a path explosion, path divergence, and constraint complexity [19]. So, using symbolic execution technique for all the program paths, the existing testing techniques have to limit their search of relevant inputs either by a depth of the symbolic path constraint, or execution time of the testing process [20, 21].

The mobile apps are highly interactive mainly because of their limited physical sizes. That is, relatively small mobile screens (comparing to the PC monitors) cannot accommodate a large number of the UI elements on one screen as we can do for PC. As such, to achieve the same desirable result on mobile, it will require more and various

kinds of user actions (e.g., it can be achieved via various user gestures [e.g., LongClick, DoubleClick, Swipe, Roll], or involving phone hardware buttons [e.g., Menu, Home, Back], or the inter-app communication). Therefore, due to the highly interactive nature of the mobile apps GUIs, they also become more state-sensitive than a PC software [2]. An internal state of a mobile app can be changed at any time depending on user actions performed earlier, or system events. Moreover, the internal state may change the behaviour of the app in certain conditions during its subsequent usage. Therefore, testing techniques which build state-aware GUI models could perform more complete/accurate testing if considering behavioural changes in the app. However, such techniques will also need to cope with a state explosion problem; otherwise, the state-aware GUI model may grow infinitely.

1.2 Objectives

In this thesis, our goal is to develop a novel automated GUI testing technique which aims to automatically generate tests with high coverage via an automated exercising of the mobile apps GUIs with relevant textual user inputs (we call them “customated” inputs). It combines the online testing technique and customated input generation while coping with the specific non-trivial GUI structure and highly interactive nature of the mobile apps. We evaluate the performance of the implemented testing technique on various real-world open-source Android apps.

To achieve our goal, we specify our objectives as follows:

- To implement a model checking technique, and an event-driven finite state machine abstraction for a systematic exploration of an app GUI. This objective arises from **Problem 1** (see above).
- To implement a customated input generation which uses UI heuristics to automatically generate relevant user predefined or random inputs and performs a symbolic execution to generate concrete user-inputs. This objective arises from **Problem 2** (see above).

1.3 Major Contributions of Thesis

In this thesis, we made two major contributions. We describe them as follows:

1. To address **Problem 1**, we introduce a novel approach to build the *f-UFG* model of the mobile app GUI. The *f-UFG* model is a finite state UI event-flow model which does not contain loops at any point in time during a testing process. The *f-UFG* model enables our testing technique to have the following benefits:

- (a) To notably reduce an overall exercising time. Our technique implements an ad-hoc GUI exploration strategy which is based on the A* search algorithm. Thus, during the exercising process, our technique can quickly find the shortest path to the interesting app UI from which the exercising process will be continued. As such, the overall exercising time is reduced which makes our technique more efficient in comparison with other automated techniques discussed in this thesis.
 - (b) To notably increase the efficacy of an automated GUI exploration. Without falling into an infinite loop, our technique explores a given app GUI-model by adding discovered app UIs one-by-one to the f -UFG model. This way, our approach ensures that all app UIs in the f -UFG model are fully explored. However, it may not discover an exact number of app UIs existing in the original app GUI-model due to the mechanism for the app UI similarity identification, and/or an automated input generation mechanism. Nevertheless, our technique shows significantly higher code coverage results in comparison with other automated techniques discussed in this thesis.
2. To address **Problem 2**, we introduce a novel approach to generate relevant textual user inputs (*customated* input generation mechanism). The customated input generation enables our testing technique to have the following benefits:
- (a) To generate concrete user inputs while mitigating well-known problems of the conventional symbolic execution. It symbolically executes only user-input-dependent statements since the textual inputs are commonly expected from a user. Our technique avoids executing all conditional statements symbolically, and thus it finishes an exercising process within a reasonable amount of time.
 - (b) To generate UI-context-aware user inputs. It uses a manually built dictionary of basic text terms which consists of various textual terms extracted from the mobile apps UIs. During an exercising process, our technique automatically obtains relevant textual inputs which lie in a set of possible values with a specific input pattern, e.g., email address, or server IP address. Therefore, our technique can automatically resolve such situations where user inputs with specific patterns are required and thus avoiding human interaction in a testing process.

1.4 Organization of Thesis

This PhD thesis consists of six chapters. Chapter 1 is an introduction to this thesis which describes the motivations, objectives, contributions, and structure of the thesis. In

Chapter 2, we give a technical background by overviewing a GUI structure, components and elements, and describe the nature of GUI testing including existing approaches, techniques and methods. In Chapter 3, we conduct a literature review, where we discuss 25 the most popular existing mobile app GUI testing tools by providing their descriptions using a four-fold approach “Introduction—Motivation—Approach—Results.” In Chapter 4, we perform automated functional testing of the mobile apps GUI. We envision this chapter as a major work where we introduce an automated GUI exerciser which aims to achieve high code coverage for the real-world Android apps. In Chapter 5, we perform an automated textual input generation for functional testing of the mobile apps GUI. We envision this chapter as an extension of our major work which is reported in Chapter 4, where we introduce an improved automated GUI testing technique which generates relevant user inputs using dynamic symbolic execution. In Chapter 6, we conclude our work and give further recommendations on how this work can be continued, and what else can be improved. In Appendix A, we perform a comparison of the Android apps disassemblers via functional testing of the mobile apps GUI, where a reader will become more familiar with the Android mobile environment, and Android GUI testing tools.

Chapter 2

Background

2.1 Overview of GUI: Structure, Components and Elements

The Android UI is a Graphical User Interface (GUI) which allows users to interact with Android devices through graphical icons and visual indicators displaying on the Android device screen. The GUI has an innate hierarchical structure, and is a graphical front-end to a software system or application which accepts as input user actions which are automatically translated into a system-predefined set of events, and produces deterministic graphical output as a result of a user input [22]. The actions on the GUI are usually performed through a direct manipulation [23–26] of the graphical elements. The direct manipulation is a style of Human-Computer Interaction (HCI) which involves continuous representation of the UI elements and rapid, reversible, and incremental actions and feedback. In HCI, designing the visual components and elements, their composition on the device screen, a behaviour of a GUI are essential parts of an application (simply “app”) development. It aims to enhance the efficiency and ease of use for the underlying logical design (business logic) of the app.

Android provides various default UI components such as structured layout objects and UI controls which allow us to design an app UI. The app UI elements are built using *View* and *ViewGroup* objects. A *View* is defined as an object which draws a visual element on a device screen, with which a user can interact. In turn, a *ViewGroup* is defined as an object which holds (contains) other *View* (and/or *ViewGroup*) objects to define an overall structure (layout) of every interface (single view hierarchy) in an app UI. Therefore, every interface in the app UI is defined using a hierarchy of *View* and *ViewGroup* objects. In Figure 2.1, we show an example of the hierarchy of the interface, where each view group is an invisible (to a user) container which organizes its child views, while the child views may be any input UI controls, e.g., buttons, text fields, checkboxes, and other widgets which draw various parts of an app interface.

Figure 2.1: A view hierarchy which defines an Android app interface structure (layout)¹

An interface view hierarchy² can be simple or complex depending on the requirements. To declare a simple interface layout, we can create *View* objects at runtime in the code and start building the layout. However, for complex interfaces, Android provides a convenient way to define a layout using XML file(s), where XML offers a human-readable structure for it.

A GUI uses a combination of visual components to provide a uniform and interactive platform for users to gather and produce information. A series of visual components conforming a visual language (a system of communication using visual components) have evolved to represent information stored on the mobile devices. The most common combination of such components in GUIs is *Windows*, *Icons*, *Menus*, *Pointer*, namely *WIMP* [27, 28]. The *WIMP* paradigm had emerged since the era of personal computing (~1960s) when personal computers (PCs) were becoming popular in everyday life. In *WIMP*, *Windows* are used to visually isolate a running program from other programs which are running at the same time in other windows; *Icon* represents a shortcut (or link) to an available action (e.g., double-click to run a program); *Menu* is a text or icon-based component which allows a user to visually select a required action for execution; and *Pointer* is an on-screen symbol which represents movement of a physical device, which a user controls to manipulate with windows, icons, menus, and other GUI components.

The *WIMP* GUIs are well-suited for PCs programs. However such interfaces are not optimised to work with complex tasks, e.g., computer-aided design, working with a large amount of data simultaneously, or interactive games. So, custom interfaces have been developed to facilitate a better encapsulation of the workspaces, actions, and objects requiring for specific complex tasks. In contrast, the *WIMP* GUIs are not well-suited for

¹<https://developer.android.com/guide/topics/ui/overview.html>

²in our work, we automatically dump (using *uiautomator*) every encountered view hierarchy of the foreground interface (device screen) into an XML-structured file, and use its information to store and restore the interface, as well as to generate relevant events, and textual user inputs for the foreground interface.

applications requiring continuous input signals, showing 3D models, or simply requiring an interaction for which there is no defined standard visual component. The interfaces which are based on such considerations, are called *post-WIMP* [29–32]. The *post-WIMP* interfaces go beyond the *WIMP* paradigm and adapt it to mobile devices such as personal digital assistants (PDAs) and smartphones. Since the era of mobile computing (~2007), the *post-WIMP* style of interaction for multi-touch screens has been popularised. It supports styles of interaction using more than one finger in contact with a display of the mobile device, which allows actions such as pinching, rotating, swiping and other unique user gestures which are unsupported by one mouse pointer. In the mid-late of 2010s, the *post-WIMP* interface on the touchscreen-based operating systems (e.g., Android, iOS, BlackBerry, Windows Phone/Mobile, Symbian) has become available to a wide public, and those devices became milestones in a development of mobile devices [33, 34].

Adapting the *post-WIMP* paradigm for the mobile apps UIs, Android provides a wide range of built-in UI components such as *Input Controls* and *Events*,³ *Menus*, *Dialogs*, *Notifications*, *Toasts*, *Adaptive Icons*, *App Shortcuts*, and others, as well as it allows a user to build *Custom* UI components by offering a sophisticated and powerful componentized model which is based on the fundamental layout classes *View* and *ViewGroup*. For example, *Input Controls* (see Figure 2.2) consist of interactive UI elements such as buttons, text fields, seek bars, check-boxes, zoom buttons, toggle buttons, and others, with which a user can manipulate an interface. On Android, each input control is added in an XML file which describes an interface layout, where each input control supports a specific set of *Input Events*, so user actions such as text entering, button touching, and others can be handled by an app. *Menus* is a common interface component in many

Figure 2.2: Example of standard Android app UI elements⁴

types of apps. On Android, Menus have several types such as *options menu*, *context*

³in our work, we use *Input Controls* and *Events* as a fundamental basis for an automated app UI testing; for every interface, we represent its UI elements as objects with their own sets of UI events so that we can manipulate the interface by mimicking actual user actions for each UI element.

⁴<https://developer.android.com/guide/topics/ui/controls.html>

menu, and *pop-up menu*. The *options menu* is the main collection of menu items which have a global impact on an app, e.g., items such as *Search*, *Compose email*, and *Settings*. The *context menu* is a floating menu which appears only when a user performs a long-click on a UI element. It provides additional actions to manipulate the selected content. The *pop-up menu* displays a list of items which is anchored to a UI element which has invoked the menu. *Dialogs* represent small windows which prompt a user to provide additional information. The dialog windows do not occupy the entire device screen and are normally used for modal events which require users to take certain action(s) before they can proceed. *Notifications* represent messages which are displayed to a user by an app while the component is outside of the app UI. The notification messages appear as icons in the system notification area. *Toasts* provide simple feedback about a user action or operation in small pop-ups. They only occupy a small portion of the mobile device screen which is required by the message, while a foreground interface remains visible and interactive. The messages automatically disappear after a certain timeout. *Adaptive Icons* allow the launcher icons to be displayed in a variety of shapes across different Android device models. For example, an adaptive launcher icon can display a circular shape on one device model, while it displays a squircle on another one. *App Shortcuts* allow us to define shortcuts to specific actions in an app, e.g., to quickly start common or recommended tasks within the app.

2.2 Nature of GUI Testing: Approaches, Techniques and Methods

A Graphical User Interface (GUI) is acknowledged as a crucial component of an event-driven software (e.g., mobile apps) [35–37]. In the event-driven software, e.g., real-world apps, the app GUI usually contains hundreds or even thousands of elements [38]. According to [39–41], a large part of the app code is dedicated to the user interface, so its testing is an essential part of the software development life cycle (SDLC),⁵ which may significantly improve an overall quality of software [42, 43]. During the testing phase, GUI can be tested by executing each event individually and observing its behaviour [44]. However, it is not a trivial task since the behaviour of an event handler may depend on a GUI internal state, state of other entities (objects, event handlers) and the external environment. Furthermore, an outcome of an event handler execution may vary depending on a particular sequence of preceding events. As a result, each GUI event needs to be tested in a context of different states via generating and executing various sequences of GUI events [5, 45].

⁵software development life cycle is the process of dividing development work into several phases to improve design, product, and project management in a cycle.

The impetus of GUI is to simplify user interaction with an app. GUI takes user actions (e.g., touches, selections, typing) as input, and changes a state of its GUI elements by translating user actions into platform-specific event handlers to execute their corresponding app functionality. Providing such the “event-handler architecture”, event handlers may be created and maintained fairly independently so that complex software may be built using these loosely coupled pieces of code while offering many degrees of usage freedom via its GUI (e.g., users may choose to perform a particular task in different ways in terms of possible user actions, their number and an execution order).

Modern mobile apps have a highly interactive nature and complex GUI structure. As such, an automated GUI testing of mobile apps is a daunting task for developers and testers. Often the GUI testing is done manually where all possible combinations of the GUI elements for a given app screen are manually tested for functional correctness and aesthetic quality. The manual GUI testing is no doubt an effective approach, however, it is inefficient, i.e., time-consuming, error-prone, and usually not exhaustive, especially for a large software with complex GUIs. So, to reduce manual efforts, or even eliminate them (it depends on an app GUI complexity), there have been several GUI testing approaches introduced such as performance testing and analysis, model-based testing⁶ [46–51], combinatorial interaction testing [52], and others. Subsequently, the GUI testing approaches can be implemented in automated GUI testing tools via different testing techniques which have been adapted to the GUI testing, e.g., Smart or Dumb Monkey testing⁶ [53–55], concolic testing⁶ [56–58], search-based testing [59, 60], and evolutionary testing [61, 62]. Depending on which testing technique is implemented, the GUI testing can be performed via dynamic⁷ or static⁸ program analysis methods or their combination.

In dynamic program analysis, an application (software) under test (AUT) must be executed with adequate test inputs (e.g., using relevant user inputs such as UI events and textual inputs) to reveal as many as possible various program behaviors and test their executions flows for correctness (e.g., input validation, unhandled and/or runtime exceptions, infinite loops and/or conditional statements). In software testing, we use code coverage⁹ as a central measure to evaluate the effectiveness of dynamic analysis. The code coverage helps to ensure that an adequate set of possible program behaviours has been observed, and their execution flows are tested. It is important to note that often,

⁶in our work, we use the model-based testing approach as a fundamental idea to perform a functional app UI testing; we implement Smart Monkey testing technique to automatically explore, build, and traverse the inferred app UI model; we implement a concolic testing technique to automatically generate relevant textual user inputs.

⁷dynamic program analysis is the method of testing software during runtime by executing it.

⁸static program analysis is the method of testing software by inspecting its code (source or object) without executing it.

⁹code coverage is a measure which is used to describe the degree of code execution completeness for the entire program once a testing procedure has finished.

dynamic analysis requires an AUT source or object code,¹⁰ or an execution environment instrumentation¹¹ to collect various attributes and properties (e.g., code coverage, memory and performance profiling information, runtime tracing information, and other interesting attributes and properties) which are decided by a test specification. However, heavy instrumentation may result in significant overhead increasing an overall time for dynamic analysis. As such, the AUT code or execution environment instrumentation should be applied minimally (e.g., to only interesting attributes and properties) for the sake of not to severely degrade overall testing performance.

In software testing, Monkey testing (or simply “Monkey”) is a random type [6,63,64] of testing techniques, where tests are carried out by a user (human, or automated testing tool) providing random inputs to entire AUT or system, and verifying its behavioural responses. Monkey objective is to check if the AUT or system crashes or hangs by providing random inputs. As such, Monkey innately does not require prior knowledge or understanding of the AUT or system so that no prior tests can be prepared. Since Monkey implements random testing, it typically starts by creating a simplified model (or without model) of AUT or system. The built model can be used to generate random inputs and/or sequences of actions. In turn, random testing can be *guided* or *unguided*. The *unguided* (undirected) random testing does not have heuristics to guide its search. The *guided* (directed) random testing extracts certain heuristics from an AUT or system to guide its search, and possibly input generation using, e.g., feedback-directed [65] or adaptive [66,67] random testing.

In Monkey testing, there are two fundamental classes of Monkeys such as Dumb Monkeys and Smart Monkeys. Dumb Monkeys are cheap to develop, fully automated, and capable of performing some basic testing. Being Monkey, Dumb Monkeys aim to break an AUT or system by finding crashes or hangs, as well as identifying bugs such as environmental and hardware failures. Dumb Monkeys do not have prior knowledge of an AUT or system, UI structure, and functional capabilities, also whether randomly generated inputs are valid or not, and are not aware of any starting and ending testing locations (points), so the entire AUT or system is new to them. For example, on Android, Monkey¹² (also known as Dump Monkey or Ignorant/Unmindful Monkey) does not require any knowledge of an app UI model, it merely generates a sequence of pseudo-random UI events such as clicks, touches, or gestures, as well as a limited number of system events. This Monkey does not consider any interface transitions and performs merely random actions at random positions on an app interface. Also, adequacy of this

¹⁰object code is a code which is produced by a compiler; it is a sequence of statements or instructions in a machine code (e.g., binary or bytecode), or an intermediate language.

¹¹instrumentation is a code insertion technique which allows us to inject specific code (probes) in AUT code (source or object), or execution environment to monitor or measure AUT performance, diagnose errors, and write trace information for runtime analysis.

¹²<https://developer.android.com/studio/test/monkey.html>

Monkey can be affected by a density and/or physical size of the UI widgets on an app interface. In addition, this Monkey is not able to generate relevant to the current app interface textual user inputs which are commonly required by mobile apps due to their highly interactive nature of UIs. To run this Monkey, a user is required to provide a total count of events to be injected into an app UI. This Monkey terminates once all events have been injected, or it has crashed or hanged an app.

Smart Monkeys¹³ ensure that an AUT or system works as expected. Smart Monkeys can be very expensive to develop especially if they are fully automated so that Smart Monkeys are usually semi-automated supporting a manual operation. An inferred model or state table generation of an AUT or system is the single major contributor to an overall cost of Smart Monkeys development. Being Monkey, Smart Monkeys also aim to break an AUT or system by finding crashes or hangs, environmental and hardware failures either manually or through performance or load testing, as well as identifying and reporting other bugs. Smart Monkeys have a brief idea about an AUT or system, know whether provided inputs are valid or not, know their own capabilities as well as capabilities of an AUT or system, know their current location in an AUT or system so that they know where they can go to find new locations, where they have already been (i.e., visited locations), and where they will be if they go by this specific path (i.e., re-visiting already visited locations to find new locations).

Model-based testing [46, 47, 51, 68, 69] is a software testing approach where a functional test model is created, and test cases are derived in whole or in part from a created model which describes expected behaviours of an application or system under test. There are two substantial activities in the model-based testing such as (1) a test model creation,¹⁴ and (2) test cases derivation.¹⁵ Depending on a model-based testing environment, created test model can be abstract, or concrete.¹⁶ The generated test cases are usually functional tests and belong to the same level of abstraction as the created test model. For example, some models may contain sufficient application or system information to generate concrete (i.e., executable) tests, and perform them immediately at runtime. However, abstract (i.e., non-executable) tests cannot be immediately executed at runtime since they are derived on an abstract level of the model. So, for their execution, abstract tests must be translated into concrete ones by mapping them to specific executable statements, or method/function calls [70]. In the case of online testing, abstract tests exist only as a concept, so they are not used as explicit artefacts. In Figure 2.3, we show an overview of the model-based testing approach.

¹³in our work, we implement an automated test driver (engine) which we classify as a Smart Monkey.

¹⁴in our work, we automatically create a dynamic graphical test model which represents (in whole or in part) the mobile app UI describing its behaviour; for the automated test model creation, we implement a model checking technique using event-driven finite state machine.

¹⁵in our work, we implement a model-based testing approach in online testing form which automatically

Figure 2.3: Model-based testing: Overview of approach¹⁷

Modelling at the behavioural level is very similar to a flow graphing. A behaviour model describes how an application or system under test responds (reacts) when certain inputs are provided, and how it behaves after time lapses (e.g., upon lapsing a certain amount of time, a system event has occurred). The system response (reaction) is therefore defined by the generated outputs which are produced for the provided inputs. In a behaviour model, sequences of actions (events) occurring in a system are usually pre-defined. However, at a specific system state, there may be more than one possible action (e.g., it may depend on the previous system state, or provided inputs, or time condition). So, most modelling techniques assume that there are multiple possible “next” actions in a system. As such, many methodologies are based on the concept of a state machine. A state machine is an example of a typical behaviour model (see Figure 2.4), where the transitions (arrows) represent the system actions (events), and the graphical icons (can be drawn in various shapes) represent the system states. The state machine operates by traversing from one state to another state. In a state-oriented modelling style, every transition contains an “Input”, and, depending on the input provided, it defines “Next State”, as well as “Outputs”, and “Actions” of a system. In turn, every state describes its possible transitions to other states, as well as actions which switch the current state to the next one. A typical example of state-based model notations is a state machine which can be represented in various forms, e.g., finite-state machine, or abstract-state machine. There are modelling techniques which support hierarchical models, i.e., when a state can be replaced by a link to another model which, in turn, defines the behaviour within the state. Hierarchical models allow describing complex system behaviours since such models can be decomposed into simpler ones by breaking down the links. Such

derives, and immediately executes the test cases at runtime.

¹⁶in our work, we create a concrete test model of the mobile app UI.

¹⁷http://web.emn.fr/x-info/atlanmod/index.php?title=Model-Based_Testing

Figure 2.4: Finite state machine: Example of system states and transitions [69]

modelling capabilities use system properties such as conditionals, or predicates, to make the transactions dependent on certain conditions (variables), or the current system context.

In model based-testing, test cases are derived from test models, and not from code, so model-based testing is usually seen as a special form of black-box testing,¹⁸ where the effectiveness of model-based testing mainly depends on the quality of a test model created (i.e., its completeness and correctness), and its test cases derivation. If a test model is machine-readable and formal (i.e., it has a well-defined structure and behavioural interpretation), a process of test cases derivation can be automated in general. There are several approaches which can be used to derive tests from a test model [49, 71–74]. However, since created test models are usually represented in a state-oriented notation (e.g., using finite state machine), a central place is occupied by the concept of a path. To derive test cases, created test state-model is traversed, e.g., by a test driver (executor), to find possible executable paths. In turn, every found executable path will derive (form) its specific test case. In general, an (executable) path can be defined as a sequence of actions (events) which allow a test driver to traverse (walkthrough) a model defining an actual use case of a system. In a path, every relevant transition and state determine what specific actions or events are required, e.g., to switch a system from its current state to the next state, or verify that an interesting us state is reached, or check that a system has responded properly to provided inputs. Once a path through the model is determined, a derived test case can be immediately executed (e.g., in case of online testing) on an actual system, or postponed until all possible paths are determined (e.g., in case of offline testing), and then executed on an actual system. When derived test cases are executed, an actual system will follow the same execution path (or sequence of the actions/events) as determined by the model path, from which test cases were derived. Such a process can be repeated for another path, which determines another use case and thus verifies another sequence of actions.

¹⁸black-box testing is a software testing approach which examines the functionality of an application or system under test without knowing details of its internal implementation, and focuses on the verification of the output which is generated as a response to the provided input; the black-box tests are primarily functional in nature, however, non-functional black-box tests may also be.

In real-world systems, not every executable path can be explored since the complexity of real systems (not trivial examples) and their created models could be extremely high. As such, the number of execution paths can be very large or even infinite (it is known as “State space explosion”). So, to derive test cases which can explore an appropriate finite number of paths within a reasonable time, test criteria (requirements and specifications) are needed to guide path selection. There are various methods can be used to guide path selection such as operational profiles, reliability and criticality data, and test coverage. However, for complex systems, model-based testing is still in an evolving field.

Concolic testing¹⁹ [56, 58, 75, 76] is a hybrid software testing technique (special type of symbolic execution, see Figure 2.5) which performs a symbolic execution along with concrete execution of a path. That is, an application or system path is simultaneously executed on concrete and symbolic values, where the generated symbolic constraints on the path are simplified by replacing their complex symbolic values with concrete ones. For example, concrete values can be used to reason concretely about complex data structures (e.g., user-defined structures), or to significantly simplify symbolic constraints when an underlying symbolic constraint solver is not capable of solving such complex symbolic values (e.g., trigonometric or user-defined functions) in the symbolic constraints. The generated symbolic constraints are further used to form respective symbolic path constraints by conjoining their corresponding symbolic constraints, and applying negation of the symbolic constraints on a path to execute ‘*true*’ and ‘*false*’ conditional branches of the symbolic constraints forming different execution paths. In turn, solving the symbolic path constraints is used to incrementally generate test inputs (i.e., concrete numeric or textual values) with the aim of maximising code coverage. To generate concrete test inputs, symbolic execution is used in conjunction with Satisfiability Modulo Theories (SMT) solvers [77, 78].

Figure 2.5: Concolic execution: Mix of concrete and symbolic (abstract) execution²⁰ [59]

¹⁹the term “concolic testing” was introduced in 2005 by Koushik Sen *et al.* in their work “*CUTE: A concolic unit testing engine for C*”.

²⁰in our work, we use concolic execution to generate concrete user inputs in numeric, and textual (string) domains; we distinguish between user-dependent and user-independent inputs, where the user-dependent inputs are generated via symbolic execution, while the user-independent inputs are generated via concrete execution (i.e., their symbolic values are replaced by concrete ones).

Symbolic execution²¹ is used as an underlying technique of concolic testing. Symbolic execution [59, 79–81] is defined as a program analysis technique which assumes that every variable in a program takes a symbolic value.²² Its goal is to systematically discover as many execution paths in a program as possible by replacing concrete values with symbolic expressions. Therefore, symbolic execution may theoretically achieve 100% of a structural coverage by exploring all possible paths in a program aiming to find real-world program bugs, but not demonstrating the correctness of the program. However, in practice, the symbolic execution has several problems such as constraint complexity, path divergence, and path explosion [19, 59], so, not every path can be explored for all but trivial programs. Furthermore, symbolic execution generally requires heavy instrumentation of a program or an execution environment which may severely degrade the overall performance of program analysis. An excellent introductory example to symbolic execution with detailed explanations is given by Roberto Baldoni *et al.* in [59].

²¹the symbolic execution technique was introduced in the mid of 1970s.

²²A symbolic value can be boolean, numerical or textual, which makes a conditional statement to be TRUE or FALSE in order to execute both branches of the statement.

Chapter 3

Literature Review

3.1 Related Work

Effectiveness and efficiency are fundamental characteristics which testing techniques are built upon, and need to be continuously improved to deliver useful solutions for researchers and practitioners. For this purpose, multiple automated techniques have been proposed and developed for mobile application GUI testing aiming to improve these characteristics. In this literature review, we analyse 25 the most relevant studies using four-fold approach “*Introduction–Motivation–Approach–Results.*”

3.1.1 Autodroid

Introduction In this work [82], the authors present a combinatorial-based testing technique for automatic generation of Android app tests. The proposed online testing technique iteratively selects and executes GUI events building the desired event sequences. It does not require a static analysis of app code (source or binary) to infer a model of a target app. Instead, the proposed combinatorial-based testing technique maintains a history of event combinations as a part of the tests generation process. It uses the historical information to greedily select and execute events, which maximises coverage. This technique enables online tests generation and execution increasing likelihood of testing app behaviours which may only occur when events are executed in specific order.

Motivation Existing techniques for automated GUI testing [2, 83–90] usually spend limited efforts for combinatorial-based testing of mobile apps. They often require static analysis of app code to build its behavioural model. It is difficult to infer an accurate model of event-driven systems (with or without app source code available). Previous research works in GUI testing of Android apps commonly use random (e.g., uniform or frequency-based) algorithms to generate event sequences [3, 10, 90, 91]. These random algorithms are based on uniform probability distribution to select GUI events, and thus

they tend to redundantly execute events without consideration of the order in which events have previously occurred. As such, the authors develop a combinatorial-based testing technique which minimises redundant execution of events while generating various unique event combinations to maximise coverage of n -way event combinations. Since so, the authors believe that this technique may be effective for testing Android app behaviours that only occur when events are executed in a specific order.

Approach The proposed online combinatorial-based testing technique requires: (1) an Android Application Package (APK) file, (2) the probability of selecting the “home” event, (3) the event combination strength of n , and (4) a test suite completion criterion. The criterion for test suite completion may be a specific number of tests required or a timeout. An event combination strength of n specifies that the technique should maximise coverage of n -event-tuples during test generation. Specificity of the combinatorial-based technique is that it maintains a set of covered event-tuples to facilitate a greedy selection of uncovered ones.

The authors implemented the combinatorial-based testing technique in a tool called Autodroid. It is built on the top of the Appium test automation framework to retrieve the hierarchy of the foreground app view. Such the hierarchy view enables Autodroid to identify GUI actions available on a screen and automatically build event abstractions. For text input fields, Autodroid fills them with text that is randomly selected from a predefined collection of strings.

Results The authors evaluate combinatorial-based technique on 10 Android apps containing from 1,026 to 7,981 lines of code each. For each app, they generate 10 test suites within two hours (120 mins). For the random, frequency-based, and combinatorial-based algorithms in Autodroid, they set the fixed probability of selecting the “home” event to 5%.

The experimental results show that 2-way and 3-way combinatorial-based tests often achieve significantly higher statement and event coverage compared to random and frequency-based tests. The results also show that there is no statistically significant difference in statement and event coverage between 2-way and 3-way combinatorial-based tests.

3.1.2 AndroFrame

Introduction In this work [92], the authors propose a fully automated black-box technique for Android apps GUI testing, namely AndroFrame. It implements the machine learning approach to explore app GUI using a reinforcement learning technique QLearning. AndroFrame automatically constructs a model of a target app and generates re-

playable test cases. Using QLearning-based exploration strategy, AndroFrame learns from a training set of apps what kinds of user actions are likely to achieve a particular goal, e.g., to detect more crashes, or increase test coverage.

Motivation Various tools have been developed for automated testing of Android apps such as PUMA [93], MobiGUITAR [2], A³E [11], SwiftHand [12], Sapienz [14], Crash-Scope [94], and DynoDroid [10]. Using model-based testing approach, these tools build a model of the target app to facilitate an automated app exploration. All these tools make a certain contribution to the state-of-the-art testing in mobile. However, random testing, Monkey [95], remains the state-of-the-practice automated testing approach. Practice shows that Monkey outperforms these advanced tools regarding the code coverage and the number of found crashes [11, 96]. However, being a powerful testing approach, Monkey has its limitations. Monkey generates tests which are practically not replayable, and thus it is hard to reproduce the discovered faults. Having large exploration state-space, machine learning technique plays a crucial role in improving the state-of-the-art automated testing which targets a specific goal. So, to address this problem, the authors propose QLearning-based testing approach [97] which combines the power of random testing while generating replayable test suites.

Approach AndroFrame explores a target app GUI based on a pre-approximated probability distribution satisfying a goal, e.g., increasing coverage or faults detection. The probability distribution of the possible GUI transitions is formed in a prioritisation matrix, called Q-Matrix, which allows AndroFrame to prioritise the transitions using QLearning.

For different training goals, AndroFrame may select actions on the screen which have a better chance of hitting a bug or increasing coverage. AndroFrame implements QLearning-based testing technique using reinforcement learning. So, in the training phase, AndroFrame explores models of apps from a training set randomly. Afterwards, it uses these models to construct a single Q-Matrix for a given goal. In the testing phase, AndroFrame explores the target app by selecting GUI actions based on the Q-Matrix.

Results To evaluate the performance of AndroFrame, the authors randomly selected 300 Android apps from F-Droid [98]. They use 200 apps to train the QLearning algorithm for both crash detection and increasing coverage. The remaining 100 apps are used for testing. The authors compare AndroFrame with Monkey, PUMA, SwiftHand, Sapienz, DynoDroid, Depth-first, and uniform random exploration strategies. The experimental results show that, on average, AndroFrame achieves the highest activity coverage with 78% when it is trained for increasing coverage and discovers the highest number of unique crashes by 13 when it is trained for crash detection.

3.1.3 Mobolic

Introduction In this work [99], the authors propose a novel automated GUI testing technique for mobile applications, namely Mobolic. It automatically generates tests with high coverage by using a combination of online testing and customised input generation. Online testing enables Mobolic to systematically explore the app GUI without falling in an infinite loop, and customised input generation enables Mobolic to automatically generate relevant user inputs.

Motivation The main challenge in GUI testing is an automatic generation of tests with high coverage. The state-of-the-art and practice automated techniques [2, 8, 10–14] fail to achieve high coverage due to two common reasons: (1) modern the mobile apps have nontrivial GUIs, i.e., their UI models are not finite state and/or contain loops in the app use-flow [1, 3]; (2) modern mobile app GUIs have a highly interactive nature, so the apps behaviours is dynamic which depends on the sensible (relevant) human inputs. Currently, none of the existing approaches addresses these two problems simultaneously. So, the authors take a challenge and develop a novel automated GUI testing technique which systematically explores nontrivial GUIs of the mobile apps, while automatically generating relevant user inputs.

Approach Mobolic is a black-box testing tool which performs a functional automated GUI testing of Android apps using an active learning approach. It builds multiple atomic concrete app GUI models and uses them as guidance for systematic GUI exploration. During testing, Mobolic also generates relevant user inputs to explore app states which are constrained by specific user inputs. Mobolic aims to generate tests with high coverage in an automated manner thus minimising or even eliminating manual efforts. Mobolic achieves its aim by taking advantage of online testing [16, 46, 100] and customised input generation.

From online testing, Mobolic inherits the model checking technique and event-driven finite-state machine abstraction to guide the testing process by dynamically building an atomic finite-state model of the app GUI without loops [101]. Using the atomic models, Mobolic explores app GUI by systematically traversing each of them. For a fast traversing, Mobolic employs an informed search algorithm A*, which is the best-known form of the Best-first search (BFS) [102]. The A* search algorithm combines BFS for efficiency with the uniform cost search for optimality and completeness. Thus, it enables Mobolic to quickly find and build the shortest path leading to the target app UI in the current model. Once the atomic model has been fully explored, Mobolic selects the next one and repeats the exploration process until all the built atomic models are fully exercised.

For customized input generation, Mobolic uses app UI (screen) heuristics and user-dependent code statements to automatically generate relevant to the current app context user inputs such as user-predefined, UI-context-aware, concrete, or random. While exploring an app GUI, specific app UIs may require particular user inputs which are relevant to that app context, e.g., user email, or exact numerical value. Practically, it is impossible to generate such inputs randomly. So, to generate relevant user inputs, Mobolic uses customized input generation mechanism. Mobolic automatically finds inputs from (1) a predefined by a user set of valid and realistic values which are defined for certain input types; or (2) the UI content itself; or (3) concrete inputs which are found by performing symbolic execution on the user-input dependent statements; or (4) random ones if sophisticated generation algorithms generated none of the inputs.

Results To evaluate the performance of Mobolic, the authors downloaded ten real-world open-source Android apps and compared it with manual testing (Human), Sapienz [14], ACTEve [8], and MobiGUITAR [2]. Our experimental results show that Mobolic achieves high code coverage within a reasonable amount of time.

For the experiment, the authors selected Human for comparison as manual testing can provide the most intelligent inputs, Sapienz as a representative of the multi-objective testing showing the best performance compared with Monkey [95] and Dynodroid [10], ACTEve as it performs automated concolic testing, and MobiGUITAR as a representative of systematic GUI exploration strategy showing the best performance compared to A³E [11], and SwiftHand [12].

For each app, the authors set a time limit (60 minutes) for Sapienz, ACTEve, and MobiGUITAR as suggested in [15]. On average, for the benchmark apps, Human achieves lower code coverage compared to Mobolic (78% vs 92%), Sapienz achieves low code coverage (53% vs 92%), ACTEve and MobiGUITAR achieve significantly lower code coverage (32% vs 92%), and (44% vs. 92%), respectively.

3.1.4 Stoa

Introduction In this work [9], the authors introduce Stoa (STOchastic model App Tester), a novel guided approach to perform stochastic model-based testing on Android apps. Stoa operates in two phases. Using an app as input, it performs static and dynamic analyses to construct a stochastic model of the app’s GUI interactions. Next, it employs Gibbs sampling to iteratively mutate the stochastic model guiding the test generation toward achieving higher code coverage, and exploring unvisited execution paths. During testing, system-level events are randomly injected to increase model coverage, and thus the effectiveness of the testing.

Motivation The mobile apps are event-centric with rich, complex graphical user interfaces (GUIs), and interact with non-trivial environments (users, other apps, devices, systems). They are typically developed in agile fashion so that only main app functionality may be tested, while the rest of the app capabilities remains untested, or tested inadequately. To tackle this challenge, many techniques and approaches [2, 8, 10, 11, 14, 103] have been proposed such as symbolic execution and concolic testing [8, 104], random testing [10, 95], evolutionary testing [14, 103], and model-based testing [105, 106]. However, random-based testing is usually incomplete and insufficient, symbolic execution, concolic and evolutionary testing is typically very slow, and model-based testing faces a state-explosion problem where the number of possible states could be infinite. In addition, previous works usually consider only UI events, while system events are ignored due to their diversity and complexity. As such, the authors propose a novel stochastic model-based testing approach to improve GUI testing of Android apps. It aims to adequately test the app functionality via the GUI model by generating various UI, and system events [107].

Approach Stocat takes an app as input and operates in two phases. First, it generates a stochastic model from the app to describe its GUI interactions. The inferred GUI model is a stochastic, and it is based on a finite state machine concept. Stocat dynamically explores app’s behaviours and constructs the stochastic model using a weighted UI exploration strategy and static analysis. Second, Stocat iteratively mutates the stochastic model and generates tests from the model mutants. Using a probabilistic approach, Stocat generates various combinations of events to exhaustively test the GUI interactions and steers testing toward unexplored execution paths. Stocat implements a guided search algorithm, which is based on Gibbs sampling, an instance of Markov Chain Monte Carlo sampling, to generate high-quality mutants, and find deep bugs. Also, Stocat enhances model-based testing by randomly inject various system-level events [108] into UI tests during Gibbs sampling.

Results Stocat was evaluated on 93 open-source apps, and compared with Monkey [95], A³E [11], and Sapienz [14]. The authors set three hours for each tool to test each app on one single emulator. Stocat was given one hour for model construction, and two hours for Gibbs sampling guided testing. The results show that Stocat covers 17-31% more code, and finds 3X more unique crashes than Monkey and Sapienz, and A³E. Furthermore, Stocat was run on 1,661 most popular apps from Google Play, and detected 2,110 previously unknown and unique crashes.

3.1.5 AimDroid

Introduction In this work [109], the authors propose a practical model-based approach for automated testing of the Android apps, namely AimDroid. It aims to explore app activities while minimising unnecessary transitions between them. AimDroid applies an activity-insulated multi-level strategy to systematically discover activities, and next, it exhaustively explores every discovered activity with a reinforcement learning guided random algorithm.

Motivation The Android apps are GUI-based, and driven by various UI- and system-level events. Due to their highly interactive nature of GUIs, it is hard to automatically test them [110–112] for all, but trivial ones. The state-of-the-practice technique such as Monkey [95] generates pseudo-random streams of UI and system events. However, it usually generates many redundant [113] and irrelevant events [114] which do not consider current app state. So, other approaches have been proposed to systematically explore app GUIs, and generate relevant inputs. Some approaches use GUI models for automatic testing, while other approaches employ symbolic execution or evolutionary techniques. However, these techniques are less scalable [15] than Monkey, and thus may not be suitable for the apps with large code size, or complex GUIs. To address these problems, the authors propose AimDroid, the Activity-Insulated Multi-level strategy to scalable, automated model-based testing of Android apps. It aims to maximise the coverage and fault detection while reducing the length of the generated event sequences.

Approach AimDroid models and explores a target app using a multi-level method described in [115]. First, AimDroid systematically explores app activities in a breadth-first manner. Next, AimDroid isolates each discovered activity in so-called “cage” and exhaustively explores it with a reinforcement learning guided fuzzing algorithm. For reinforcement learning, AimDroid uses SARSA [116]. It enables AimDroid to predict and select events which are more likely to discover yet unexplored activities or crashes in a greedy manner. AimDroid splits the overall testing process into episodes. In each episode, AimDroid generates a limited number of events for each discovered activity while disabling their transitions. Therefore, it allows the testing process to reduce the overall search complexity, many app restarts and exploration time.

Results The authors conducted an empirical study on 50 real-world apps using physical Android devices. They compared the performance of AimDroid with Monkey [95] and Sapienz [14]. For each tool, it was given one hour [14] to test each app on one Android device. If a tool has finished the testing process before the given timeout, it restarts the process to continue testing.

For each tool, the authors report the activity, method, instruction coverage, and the number of unique crashes for each app. The authors also indicate that they report the results which were collected within one hour. For the activity coverage, AimDroid significantly outperforms Sapienz and Monkey. It shows the effectiveness of the activity-insulated exploration strategy. For the method and instruction coverage, AimDroid also has better results for specific apps. However, it is not as significant as the activity coverage. For crashes, AimDroid triggered 80 unique crashes, while Sapienz and Monkey triggered 16 and four unique crashes, respectively. The uniqueness of each crash was identified by an error message and a crashing activity name.

3.1.6 EHBDroid

Introduction In this work [117], the authors present a novel approach for testing Android apps, namely EHBDroid. In contrast to the conventional GUI testing approaches, EHBDroid directly invokes event handlers so that it does not need to generate complex UI or system events, and also it eliminates latency induced by the rendering of app UI screens during their transitioning.

Motivation Automated testing of mobile apps is a challenging task. There are two main aspects. First, the GUI exploration space could be unlimited, i.e., there could be an infinite number of unique GUI states yielded by UI and system events [10]. So, it is impractical to generate all possible events and their permutations within a reasonable amount of time. Second, various UI events (e.g., drag, hover), inter-app, and system events are difficult to generate in an automated manner. In addition, certain UI events can only be successfully triggered if their preconditions are satisfied (e.g., inputs to all the relevant widgets are provided) [87]. The probability to generate all combinations of the possible inputs is very small [118], so that it is hard to mimic such UI events. For many system events, proper data (e.g., arguments) needs to be constructed and dispatched correctly to the app. For inter-app events, they can only be triggered by external apps under certain conditions. As such, for many systems and complex UI events, it is much easier to invoke the event handlers directly rather than generate events from the UI or system.

There have been multiple studies [8, 10–14, 79, 87, 115, 119–121] conducted for mobile app testing. However, they are still unsatisfactory. According to [15], most of them infer a model from app GUI so that they are either too slow or cannot effectively generate complex events. Due to such reasons, the authors are intended to develop an efficient and effective automated testing technique which ensures correctness, reliability, and security of mobile apps.

Approach EHBDroid is an Event Handler-Based automated testing technique for Android apps. EHBDroid is was adapted to event-driven systems, where often exists a correspondence between an event (system, or user), and an event handler (method processing the event). On Android, for example, the UI events are handled by their corresponding event handlers, e.g., callback functions [122]. Hence, instead of generating complex UI events, EHBDroid directly triggers their event handlers. Using direct invocations of the event handlers, EHBDroid can reduce testing time since it eliminates the latency induced by GUI. Also, it can test those callback functions which cannot be easily invoked through the GUI. As a result, EHBDroid provides high code coverage in short testing time. Also, the events are systematically generated and are not redundant.

Results We have evaluated EHBDroid on a collection of 35 real-world Android apps from F-Droid [98] and Google Play, and compared it with two popular UI-based app testing approaches: Monkey [95] and Dynodroid [10], both of which have proven fault detection ability [15]. We use statement coverage as the criterion to compare the testing the effectiveness of different approaches.

The authors evaluated EHBDroid on 35 real-world large-scale Android Apps from F-Droid [98] and Google Play, and compared its performance with two state-of-the-art approaches, Monkey and Dynodroid. The experimental results show that EHBDroid is significantly more effective and efficient. It can quickly reach higher statement coverage as well as detect more bugs in a shorter time. EHBDroid achieves, on average, 11.1% higher statement coverage, and found 12 bugs in the testing apps, including five unique bugs which were not found by Monkey and Dynodroid.

3.1.7 PATDroid

Introduction In this work [123], the authors present an automated permission-aware GUI testing technique for Android apps, namely PATDroid. It aims to efficiently explore Android app functionality while considering its permissions, and behavioural changes depending on the Android app permissions. PATDroid performs a hybrid program analysis of a target app and its test harness to identify which tests should be executed on what permission combinations.

Motivation A dynamic permission system in Android (since API level 23) allows the users to grant and revoke permissions after app installation. Tests which are executed on an app may pass under certain combinations of the granted permissions, while the same tests may fail under different combinations. So, the dynamic permission system makes automated testing more challenging since an app behaviour may change depending on the permissions. As such, the app needs to be tested under a vast range of permission

combinations. However, such the number of permissions permutations is usually large, and even automated testing is still time-consuming for such testing, and thus often impractical. To address this problem, the authors propose an automated permission-aware GUI testing technique for Android apps.

Approach PATDroid uses a hybrid program analysis approach to find the dependencies between an app’s GUI tests and their permissions. It first identifies the entry-points of the app exercised by each test case. Next, it examines the app code which is reachable from the identified entry points and finds which code portions are permission-protected. Afterwards, it identifies GUI widgets which can trigger the permission-protected code. Finally, it performs linkage of GUI widgets in the test scripts. Thus, PATDroid employs a relatively precise, and yet scalable, automated testing technique, which can effectively construct specific tests which can execute permission-protected code segments.

Results The experimental results show that PATDroid significantly reduces the overall testing time, while achieves reasonable code coverage and fault detection rate. On average, PATDroid reduces the total number of tests and the execution time by 71%, while providing comparable code coverage as standard tests running for all permutations of permissions. In addition, PATDroid was able to identify several bugs in the real-world apps which can only be revealed using a specific permission combination.

3.1.8 Xdroid

Introduction In this work [124], the authors present Xdroid, a tool which implements a semi-automatic testing technique combining the benefits of automated and manual testing. It aims to improve test coverage and bugs detection rate by effectively exploring different app states. Xdroid identifies when and what dependencies apps require, and injects them correspondingly in a lightweight manner. The dependencies injection is based on the fact that most of the apps dependable resources can be generated automatically or resolved manually.

Motivation Existing testing tools mainly focus on generating test inputs by exploring apps functionalities with different strategies. However, the state-of-the-art tools give low coverage for certain kinds of apps [15]. This is because most of them depend on various external resources (e.g., files, system instances). So, without providing such necessary app dependencies, it is hard to generate effective test inputs, and adequately explore app states. Such external dependencies are the critical obstacle which obscures the efficiency of the test automation. To address this problem, the authors facilitate existing testing

tools by developing Xdroid which helps to resolve the apps dependency shortage problem in an automated manner.

Approach To efficiently explore different states of the app, Xdroid runs in a separate thread forked by the target app process. Xdroid works as a middleware which lies between the Android OS and host app. Xdroid uses Xposed framework from Xposed Module Repository to intercept relevant methods, and thus to monitor and inject dependencies when resource shortages are detected. Xdroid resolves the app dependencies shortage problem using two mechanisms: injection and notification. For the dependencies which can be generated automatically, it injects predefined contents and files to the system. For the dependencies which require human intervention, it notifies the user by asking for manual inputs. To reduce manual efforts, Xdroid stores the provided user inputs in the database for future reuse in the tests generation.

A recent study [15] shows that Monkey [95] has the best performance by multiple criteria. So, based on Monkey, Xdroid implements its Xmonkey sub-module to automate the testing process while keeping the capability of switching between manual and automated modes. Using the built-in Xmonkey, Xdroid implements an effective test engine which explores various app states, reveals deep bugs, and increases the overall test coverage.

Results The authors evaluated the performance of Xdroid on 20 real-world and compared it with three prevalent approaches: manual testing, Sapienz [14], and Monkey [95]. The results show that, on average, Xdroid outperforms Monkey on 17%, and Sapienz on 22% regarding code coverage, and also reveals more bugs than manual testing.

3.1.9 LAND

Introduction In this work [125], the authors propose a widget-sensitive and back-stack-aware automated GUI exploration approach for testing Android apps, namely LAND. LAND dynamically builds a model from the target app GUI, namely LATTE (Labeled Transition graph with sTack and widgEt). The LATTE model maintains the GUI widget states and back-stack information at runtime. To create an accurate yet compact model, LAND balances the LATTE model by introducing a metric “state similarity” thus similar app states can be merged into single one.

Motivation Various automated model-based testing approaches for Android apps have been proposed: static [87, 126], and dynamic [12, 13] ones. In practice, static techniques fail to capture the dynamic behaviour of an app, which vary depending on the user inputs; dynamic techniques focus on observable app behaviour during runtime while missing platform-specific characteristics such as “back stack”. However, widgets state

(app property), and “back stack” (platform property) are important for precise model construction and adequate testing of mobile apps. So, to tackle this challenge, the authors implement an approach in tool LAND which constructs widget-sensitive and back-stack-aware model LATTE.

Approach LAND is built to answer three main questions: (1) how to automatically obtain accurate back-stack information during runtime? (2) how to automatically extract GUI widgets states and construct an accurate LATTE model? (3) how to systematically traverse a built LATTE model and explore a target app. To answer these questions, LAND implements four modules: (1) “Apk Instrumentation”, (2) “Monitor”, (3) “Model Constructor”, and (4) “GUI Explorer”. “Apk Instrumentation” analyses the app byte-code and inserts probes (portions of code) for logging app runtime information. “Monitor” collects information about the GUI widgets status and their corresponding executed code. “Model Constructor” builds an initial LATTE model from the information collected by “Monitor”, and further refines the model during runtime. “GUI Explorer” is a test driver which determines which event to be triggered next (e.g., click a button).

Results To evaluate the performance of the LAND, the authors obtained 20 real-world popular Android apps, and compared it with Monkey [95] and Dynodroid [10]. Using the LATTE model, LAND shows that it is capable of exploring more app states than Monkey and Dynodroid. The experimental results show that, for the benchmark apps used, LAND achieves 20% higher code coverage than Monkey and Dynodroid, on average.

3.1.10 DroidWalker

Introduction In this work [127], the authors present a dynamic-analysis technique, namely DroidWalker, to automatically generate re-playable test cases for Android apps. The main idea of this technique is to dynamically build the adaptive model which accurately reflects the app states so that every state can be re-visited individually. DroidWalker systematically explores the target app to construct the model. Afterwards, the users can choose which state in the model should be reproduced, and DroidWalker executes the generated test cases to reach the target app state.

Motivation Generating test cases via the automatic exploration of the app model is a promising approach. Many researchers and practitioners have proposed and implemented various automated exploration techniques for Android apps. However, none of them can generate re-playable test cases [15]. The authors highlight the importance of such test cases. The re-playable test cases aim to reach the same app state when the test case is re-executed. So the re-playable test cases can be used for the failures reproducing,

ensuring correctness of the code changes via regression testing, and verifying an app behaviour consistency across different Android platforms. To address this problem, the authors develop DroidWalker tool to generate re-playable test cases.

Approach DroidWalker uses a model-based approach to systematically explore app states and construct a state model. It builds a model from app GUI using a finite-state machine paradigm. DroidWalker uses the app UI structure to identify similar app states. It is based on the fact that the UIs which are rendered by the same app behaviour share the same UI structure. For each app state, DroidWalker extracts executable widgets (which the user can interact with), and systematically generates the relevant events for each of them. The executable widgets may bring the app to another state. So, if a new state is identified, DroidWalker applies the exploration algorithm on the new state. In such a way, DroidWalker systematically explores all app states. When the exploration on the app state is complete, DroidWalker performs backtrack to return to the previous state and continues exploring from there. Once the model has been fully explored, the user selects a target state from the model, and DroidWalker traverses the model in a depth-first manner and generates event sequences leading the testing procedure to the target app state.

Results The authors evaluated the performance of DroidWalker on 20 real-world Android apps, and compared it with A³E [11], GUIRipper [13], and PUMA [93]. As suggested in [15], for each app, they run the test for 60 minutes. The results show that DroidWalker achieves higher line coverage than A³E, GUIRipper and PUMA, and also reveals more bugs.

3.1.11 Sapienz

Introduction In this work [14], the authors introduce a novel testing technique for automated testing of Android apps. It uses multi-objective search-based algorithms to automatically explore apps, and optimise tests performance by minimising the length of the generated event sequences while maximising code coverage and fault detection rate. It combines uniform random, systematic and search-based exploration strategies, and uses random string seeding and multi-level instrumentation.

Motivation In mobile, app testing still mainly relies on manual testing [128], while the use of automated techniques is rare [129]. For the automated testing, practitioners usually use default Android Monkey [95]. This tool is considered as the state-of-the-practice for automated software testing [103]. However, Monkey dumbly automates testing. It simply generates from short to long event sequences in a random manner hoping to maximally

explore the target app, and reveal multiple app failures. Due to its random nature, Monkey may require long event sequences to reveal actual app failures. Unfortunately, the long sequences are likely to be impractical. Another automated technique such as exploratory testing [130] can be effective and thus practical [131–133]. However, for mobile app testing [134, 135], it is still an evolving technique. To address these limitations, the authors introduce a multi-objective automated testing technique for Android apps, namely Sapienz.

Approach Sapienz adapts and extends the traditional automated testing techniques for mobile testing. Sapienz combines uniform random, systematic and search-based exploration strategies, uses random string seeding and multi-level instrumentation. The main idea of Sapienz is to implement a fully automated testing approach which generates shorter event sequences while maximising fault detection rate and code coverage. Sapienz is the first automated exploratory testing technique which uses Pareto multi-objective approach to Android apps testing. By employing Pareto-optimal multi-objective search-based approach, Sapienz generates long event sequences when only they can reveal faults, or must be executed to achieve higher code coverage. So, with Pareto optimality, Sapienz can safely replace such long event sequences with shorter ones when they are of equivalent quality.

Results To evaluate the performance of Sapienz, the authors selected 68 apps from an Android benchmark suite [15], and compared it with Dynodroid and Monkey. The results show that Sapienz outperforms Dynodroid and Monkey for the progressive code coverage within 60 mins. For the second dataset of 10 random apps from F-Droid [98], Sapienz outperforms Dynodroid and Monkey in 7/10 experiments for coverage, 7/10 for fault detection and 10/10 for fault-detection event sequences. Proving its practicality and usefulness, from 1,000 Google Play apps, Sapienz found 558 previously unknown crashes.

3.1.12 GUICC

Introduction In this work [115], the authors focus on the GUI modelling technique developing the GUI comparison criteria for model generation. This modelling technique defines a multi-level GUI Comparison Criteria (GUICC) which suggests several levels of abstraction for constructing a precise GUI model for an automated GUI testing. The proposed GUICC is envisioned as a set of rules which are used to compare two GUIs, i.e., if they have an equivalent state or different one.

Motivation Many model-based GUI testing techniques have been developed to facilitate the systematic GUI exploration [13, 136]. The model-based techniques usually build a behavioural GUI model and generate test inputs based on the inferred model [137, 138]. However, for effective model-based GUI testing, building a precise model is crucial to generating effective test inputs [12]. A GUI model which contains small exploration space can have poor the testing effectiveness, and dynamic behaviours in GUIs can cause inconsistent model generation or state explosion problems due to the non-deterministic nature of their behaviours. So, the authors tackle the behaviour modelling problems of real-world apps which is a key insight of an effective model-based GUI testing. They specify the level of behaviour abstraction by defining 5-level GUI comparison criteria.

Approach The GUICC determine the equivalence or difference between two GUI states. During execution, GUICC compares a GUI of the current screen on Android device with next one. By comparing GUIs with GUICC, the testing tool can determine how to update the GUI model. Based on the empirical investigation of real-world apps, the authors designed multi-level GUICC criteria which comprise of hierarchical relationships between GUIs. The comparison criteria have five levels (C-Lv), and it has three types of outputs according to the comparison result: T (Terminated), S (Same), or N (New). The default comparison level is set to C-Lv5. However, it can be adjusted according to the desired level of abstraction. As C-Lv increases from C-Lv1 to C-Lv5, more concrete (i.e., fine-grained) GUI information is used to compare GUI states.

- C-Lv1: Compare Package Names. It compares the package name of the currently running app screen with visited ones.
- C-Lv2: Compare Activity Names. It compares the activity name of the currently running app screen with the names of the visited ones.
- C-Lv3: Compare Widget Composition. It compares app screen widgets hierarchy obtained with the aid of UI Automator [139].
- C-Lv4: Compare Executable Widgets. It compares executable widgets between GUI states using a comparison technique for event handlers [12].
- C-Lv5: Compare Contents. It compares app screen contents information which is obtained by extracting values of UI elements attributes: <text>, and <content-desc>.

Results The authors analysed the performance of GUICC from three aspects: GUI model generation, code coverage, and error detection. They randomly selected 10 medium-sized (1,000–10,000 LOC) apps for their evaluation from [15]. For GUI modelling, the authors observe that GUICC can vary depending on the behaviours of an AUT,

so GUICC must be carefully selected considering the target app. Also, higher levels of GUICC usually result in larger GUI models. However, higher-level GUICC do not always perform better due to the detailed comparison which can be disruptive for efficient testing.

For code coverage, the authors observe that GUICC can affect the code coverage. Increasing the C-Lv usually increases the code coverage for all benchmark apps due to their larger inferred GUI models. For error detection, the authors observe that the runtime errors were only revealed in the fine-grained GUI graphs with C-Lv4 and C-Lv5, while the lower level of GUICC could not detect them. This observation shows that setting the level of behaviour abstraction affects not only the code coverage but also the error detection ability.

3.1.13 DroidDEV

Introduction In this work [101], the authors present a novel automated UI testing technique, namely DroidDEV, to achieve high code coverage for the real-world Android apps. DroidDEV is an effective, and yet reasonably efficient automated testing technique. For the effective app UI exploration, it dynamically builds multiple finite-state UI flow graphs of the target app UI model and generates UI-context-aware textual inputs. Thus, it enables DroidDEV to systematically explore each of the built UI graphs in a user-like manner. For the efficient app UI exploration, DroidDEV implements one of the greedy search algorithms such as Best-first search, an informed search algorithm, to find the shortest UI path leading to the target UI.

Motivation Innately, Android apps and their UIs are crafted with a mindset of being used by humans, not machines. As such, Android apps usually have a non-trivial and highly interactive user interface. This fact makes automated testing a challenging task. Practice shows, that existing state-of-the-art automated UI testing techniques for Android apps [2, 10–13, 88, 119] still cannot adequately exercise apps UIs [2, 3], and thus provide a low code coverage [15]. The authors identify two main problems why existing automated UI testing techniques poorly exercise Android apps: (1) Android apps have non-trivial UI models, and (2) Android apps functionality rely on sensible (humanoid) user inputs. To address these limitations, the authors develop DroidDEV (AnDroid Dynamic, Exerciser, high coVerage), to achieve high code coverage, and automatically generate sensible textual user inputs for the mobile apps.

Approach To exhaustively explore all app UIs, DroidDEV dynamically builds multiple finite-state UI flow graphs to facilitate automatic app UI exploration which is guided by the Best-first search. DroidDEV systematically explores app UI by exhaustively

generating relevant UI events, and UI-context-aware textual inputs to discover deep app UIs. DroidDEV identifies itself by the following key features: (1) self-terminative, (2) automated, (3) effective, and (4) efficient. DroidDEV implements a self-terminative exploration procedure, and thus it ensures that there is no loop in the constructed finite-state UI flow graphs. DroidDEV is an automated testing technique, and thus it generates relevant UI-context-aware textual inputs to fully automate the testing process. DroidDEV is an efficient testing technique, and thus it implements the informed Best-first search algorithm to quickly reach the target app UI through the shortest UI path. DroidDEV is an effective testing technique, and thus it systematically explores app UI by exercising all the widgets on all the discovered app UIs.

Results The authors evaluated the performance of DroidDEV on 20 real-world Android apps and compared it with manual testing, MobiGUITAR and Monkey. The performance was evaluated regarding the achieved code coverage, using basic-block as a coverage metric, and exercising time. The experimental results show that, on average, DroidDEV and manual UI testing perform similarly for the code coverage (91.5% vs 91.0%, respectively). However, on average, DroidDEV significantly outperforms MobiGUITAR and Monkey for the code coverage (91.5% vs 44.9% and 71.9%, respectively), while taking shorter exercising time (15.76 mins vs 60 mins).

3.1.14 MCrawlT

Introduction In this work [140], the authors propose an automatic testing technique for mobile applications, namely MCrawlT. It automatically learns an app GUI model through its transitions and app states. The authors mainly focus on the quality of the inferred model. MCrawlT constructs formal and exact models which are dynamically built via automatic exploration of the app GUI. While inferring a precise model, MCrawlT avoids the state-space explosion, a common problem in automated model-based testing. During execution, it applies state-equivalence criteria to reduce the state-space search, and thus ensuring that it always has a finite reachable state.

Motivation Manual testing is the de-facto most effective testing approach. However, manual testing is often error-prone and time-consuming. Multiple techniques have already been proposed to automate apps GUI testing, e.g., desktop applications [22], Web-applications [141] or mobile ones [8]. These techniques explore a target app aiming to obtain high code coverage within a short time. So, due to the time limit, such automated techniques often learn an approximate model of the target app, which is usually sufficient for guiding the testing process. However, the precision of the learned models is an important criterion. Approximate models could be incomplete leading to false positives,

such as unexpected app crashes or failures which are not related to a target app. To address this problem, the authors develop a technique aiming to learn an exact model of a target mobile app.

Approach MCrawlIT implements an algorithm which builds an exact app model which is extracted from GUIs states, screen content, and app notifications. The inferred model is called Parameterised Labelled Transition System (in short, PLTS). PLTS captures all generated events and the widget properties which are extracted from the explored screens, as well as the app notifications. All such data gives a comprehensive description of the behavioural app model which is exposed during a testing process.

During the model construction, the testing process may find the infinite number of app states due to a possible state-space explosion. So, to mitigate such limitation, the authors consider the notion of state abstraction [138, 142] which is formally defined to limit the exploration state-space. Using the defined state-similarity criteria, MCrawlIT algorithm dynamically refines the learned model according to the current app context. Thus, MCrawlIT reduces the exploration state-space by merging similar states since they are not required to be re-explored multiple times. Without loss of generality, the exact learned model can also be used to detect various app bugs and crashes, and generate re-playable test cases to reproduce faulty app states.

Results To evaluate the performance of MCrawlIT, the authors obtained 30 Android apps, and compared it with Monkey [95], ORBIT [119], GUITAR [22], SwiftHand [12] and Dynodroid [10]. For each tool, the authors set three hours time limit for all 30 apps. The results show that MCrawlIT achieves significantly higher code coverage than the other tools within a given time frame. On benchmark of 30 apps, MCrawlIT provides better code coverage results which lie in the interval 25%–96% and outperforms the other tools in automatic crash detection.

3.1.15 GAT

Introduction In this work [143], the authors present a static analysis technique which extracts the gesture information of the UI components. The static analysis aims to reduce the number of gestures to be dynamically delivered during a testing process. The authors implemented the proposed technique in a gesture-guided event generation tool, namely GAT (Gesture-Aware Testing).

Motivation Gestures are the common means of human interaction with mobile devices. However, human gestures hinder GUI testing making it difficult for automated testing. Due to the pressing time to market, mobile apps are often poorly tested. Manual testing

is time consuming, and automated testing is not effective enough to rely upon [8, 10–13, 144, 145]. The existing automated testing techniques do not focus on gesture event generation. Instead, they focus on a small subset of gestures or considers only simple ones, e.g., touch gestures. To address the problem of automatic gestures generation, the authors propose a gesture-guided testing technique which facilitates effective, and yet efficient automated testing of gesture-intensive mobile apps.

Approach The main challenge of this approach is to obtain relevant gestures for each UI component. The authors address this challenge by enhancing the existing GUI model-based testing technique [11] and using static analysis tool Soot [146]. GAT extracts relevant UI components gestures and uses them to explore the target app by guiding the automated testing process.

To implement an effective and efficient testing technique, the authors perform three main steps: (1) Study of Gesture Recognition in Android apps, (2) Extracting Gesture Information, and (3) Integrating Gesture Information in Event Generation. In the first step, the authors conduct an empirical study on several Android apps to find the categories of gesture recognition, which demonstrates how to relate the UI components with their relevant gestures. In the second step, for each category of gesture recognition, the authors perform static analysis and execute an algorithm to extract the corresponding gesture information for each UI component. In the third step, the authors integrate the extracted gesture information into the testing process and thus exercise the respective UI components with their relevant gestures.

Results To evaluate the performance of GAT, the authors ran it on nine Android apps and compared it with (1) a technique which generates only simple gestures, (2) a technique which exhaustively explores all possible gestures, and Monkey [95] which generates random gestures. Evaluation results show that GAT has the similar capability of detecting real-world gesture-related bugs as the second technique (2). Also, GAT outperforms the first technique (1), and Monkey in covering more gesture-related code. At last, GAT achieves the same code coverage as the second technique (2), but in a shorter time.

3.1.16 MobiGUITAR

Introduction In this work [2], the authors introduce a technique which automates GUI testing of Android apps, namely Mobile GUI Testing Framework (MobiGUITAR). It extracts an abstraction of GUI widgets state during run-time forming a state-aware model. The extracted model provides the guidance to automatically generate test cases with high coverage.

Motivation The previous authors’ work [13] shows the effectiveness of using model-learning approach for automated testing of Android apps. Their previous model is based on the event-flow graph (EFG) [147], which is stateless. The stateless model helps to mitigate the state-space explosion problem. However, using a stateless model is no longer adequate since the modern mobile apps are highly state-sensitive. Furthermore, the originally proposed test adequacy criteria based on EFGs cannot be used “as is”. So, the new state-sensitive criteria are needed. Automatic test drivers which are based on EFGs, are also unusable. As such, the authors need to develop new state-sensitive criteria and automatic test drivers which operate on state machines. To address these problems, the authors introduce MobiGUITAR which performs state-aware automated testing of Android apps.

Approach Although MobiGUITAR uses similar concepts which are proposed in the context of desktop Java apps [148], and Web applications [142, 149], it implements new test adequacy criteria adapted to the state machines. MobiGUITAR uses the state-aware models and test adequacy criteria to generate test cases automatically. MobiGUITAR works in three steps: (1) ripping, (2) generation, and (3) execution. In the ripping step, MobiGUITAR dynamically explores an app GUI and creates its state-aware model (not an EFG). Because MobiGUITAR is fully automated, it creates the model via automatic GUI ripping. MobiGUITAR extends its previous technique AndroidRipper [13]. In the generation step, MobiGUITAR uses the built model and test adequacy criteria to generate test cases which are written as sequences of possible GUI events. For the test adequacy criteria, the authors introduce a pairwise edge coverage criterion meaning that all generated events must be exercised in pairs. In the execution step, MobiGUITAR replays the generated tests. MobiGUITAR outputs test cases in the JUnit format so that a user can modify the test cases according to the test requirements.

Results For the MobiGUITAR evaluation, the authors selected four apps from Google Play, namely Aard Dictionary, Tomdroid, Book Catalogue, and WordPress. To show MobiGUITAR bug detection capability, the authors automatically executed MobiGUITAR on the benchmark apps, generated and executed 7,711 test cases. These test cases revealed ten bugs: seven bugs were detected during the ripping step, and three bugs were new ones. The authors also compared MobiGUITAR with Monkey [95] and Dynodroid [10]. Both tools found only three of 10 bugs and failed to find any other bugs. The overall experimental results show that MobiGUITAR generates such test cases which can reveal critical bugs in the apps. Furthermore, the authors show that model-learning and model-based testing is a promising combination for detecting deep bugs in the mobile app testing.

3.1.17 SIG–Droid

Introduction In this work [145], the authors present a framework which combines model-based testing and symbolic execution, namely System Input Generation (SIG–Droid). It performs automated testing of Android apps using dynamically built app models and symbolic execution. The main focus of this work is on the generation of system-level inputs for Android apps. The authors aim to generate test inputs which ensure high reachability of each executable program path. For the automated testing, SIG–Droid infers two models: Interface Model and Behavior Model. The Interface Model describes external app interfaces (rendered screens), and the Behavior Model describes the event-driven behaviour of the app including events and their handlers.

Motivation Tremendous efforts have been done in automated testing of Android apps. However, random testing, Android Monkey [95], Dynodroid [10], remains the prevalent approach which is currently state-of-the-practice. Monkey is a default Android fuzzing testing tool which generates random events, gestures, and the limited number of system-level events. Dynodroid improves random testing by employing a feedback-directed approach. Other automated techniques [3,8,12,13,119] mainly focus on testing of the apps through their GUI models by mimicking user actions, while generation of the system-level events is commonly avoided due to their high complexity, and variations depending on the Android platform. System-Level events commonly depend on data values provided by the users via app interfaces. So, to address the problem of complex inputs construction, the authors propose an automated system for system-level input generation which leverages dynamic program analysis techniques with symbolic execution [150].

Approach SIG–Droid combines program analysis techniques with symbolic execution to systematically generate inputs for Android apps. SIG–Droid uses an app source code as input and extracts two models: Behavior Model, and Interface Model. The Behavior Model describes the app behaviour via its event handlers sequences, and the Interface Model describes the app interface hierarchy. These models are used to guide the testing process by mimicking user actions on the app interface. In particular, SIG–Droid uses the Interface Model to retrieve input values which an app receives through its editable UI widgets, and it uses the Behavior Model to drive the testing process. SIG–Droid finds execution paths which depend on the app interface inputs, exchanges concrete input values with symbolic ones and collects constraints for the interface inputs. After resolving symbolic constraints, the solved (concrete) values are used along the execution path with the corresponding UI events to discover other possible execution paths.

Results The authors evaluated SIG–Droid on several real-world apps which are downloaded from F-Droid [98]. They manually selected apps from various categories with

a criterion being such that the apps must be built using standard GUI widgets (from Android framework). The authors measured the code statement coverage and compared it with Monkey [95], and Dynodroid [10]. The results show that SIG-Droid achieves higher code coverage for all benchmark apps. On average, it outperforms Monkey and Dynodroid by 57% and 41%, respectively. Also, SIG-Droid is more efficient running 2X faster than Dynodroid. However, it is 3X slower than Monkey which is explained by its purely random nature.

3.1.18 EvoDroid

Introduction In this work [103], the authors present EvoDroid, an evolutionary testing approach for automated testing of Android apps. Evolutionary testing is a form of search-based testing, where test cases are generated according to certain heuristics to maximise the code coverage. EvoDroid combines two novel techniques: (1) an Android-specific program analysis technique that identifies the segments of the code, and (2) an evolutionary algorithm which performs a step-wise search for test cases reaching deep into the app code.

Motivation The research was inspired by previous work [151] that has shown evolutionary testing to be effective when sequences of method invocation are important for obtaining high code coverage. However, evolutionary testing techniques [62, 151–153] are typically limited to local or unit testing, as when applied at the system level, it cannot generate high-quality mutants of the test cases during the search.

Approach EvoDroid requires an Android app’s source code as input. It statically analyses the app code and infers a model of its behaviour. The model captures (1) the dependencies among the app code, and (2) the entry points of the app (i.e., places in the code that the app receives external inputs). The inferred model is used to determine how the generated test cases should be mutated for next tests generation. In particular, from the source code, EvoDroid extracts two types of models to automatically generate the tests. One model represents the app’s external interfaces, and another one represents internal app behaviours: Interface Model (IM) and Call-Graph Model (CGM).

IM provides a representation of the app’s external interfaces and particular ways in which an app can be exercised, e.g., the inputs and events available on various screens to generate tests that are valid for those screens. CGM is an extended representation of the app’s call graph. Using these two models, EvoDroid employs a step-wise evolutionary test generation algorithm, which aims to generate test cases covering as many unique CGM paths as possible. The evolutionary process is continued until all of the paths and their segments are covered or a configurable threshold (e.g., time limit, a certain level of

code coverage, the number of total test cases) is reached.

Results The authors compared EvoDroid with Monkey [95], and Dynodroid [10] in terms of code coverage and execution time. They selected ten open source apps to evaluate the line coverage between EvoDroid, Monkey, and Dynodroid. Results show that EvoDroid consistently achieves significantly higher coverage than both Monkey and Dynodroid. On average EvoDroid achieves 47% and 27% higher coverage than Monkey and Dynodroid, respectively. Also, the results demonstrate that EvoDroid (as well as any other evolutionary testing approach) performs poorly in cases where the apps are highly constrained (e.g., the probability of satisfying many conditional constraints with random inputs is close to zero, such as an if-condition that specifies an input value to be equal to a specific value).

3.1.19 ADAutomation

Introduction In this work [154], the authors develop an automated GUI testing framework for mobile applications, namely ADAutomation. It supports user behaviour modelling and extraction, and automated GUI tests generation. For model building, ADAutomation uses the language of UML activity diagrams. Based on the built models, it implements a practical approach for automated GUI tests generation. Itself, ADAutomation is implemented as a toolchain which automatically performs GUI testing and debugging.

Motivation An automated GUI-testing requires accurate modelling of the user actions to mimic the interactions in a user-like manner. Currently, state-of-the-art GUI testing approaches and tools [4, 13, 148, 155, 156] are mainly based on the event-flow graphs using a finite state machine as an underlying concept. Although such a modelling concept is promising in the automated GUI-testing, it cannot accurately reflect all possible user behaviours naturally and compactly. Therefore, it is hard to guarantee the completeness and adequacy of the generated tests which are derived from such models. To address this problem, the authors exploit the Unified Modeling Language (UML) which is a defacto modelling language describing all possible user actions [157]. So, using the UML activity diagrams, the authors propose the automated GUI testing techniques which support user behaviour modelling, GUI test case generation, and post-test analysis and debugging.

Approach ADAutomation runs in five steps: user behaviour modelling, test script library construction, GUI test case generation, test case execution, and post-test analysis. ADAutomation uses UML activity diagrams which are derived from the GUI specifications. In the UML activity diagrams, the user behaviours are defined as sequences of

possible GUI actions. So, ADAutomation attempts to extract all possible user behaviours and coverage criteria for the test adequacy evaluation. To enable GUI testing, ADAutomation uses XMI parser to obtain the GUI widget layout information (e.g., position, label), and the action-function mapping. Next, the extracted GUI information is used to construct a test script library (i.e., a set of GUI test functions). Based on the extracted user action sequences, the test script generator maps user behaviours into their respective GUI actions and writes them in the test cases. Afterwards, the generated test cases are executed on the emulator, and execution logs are recorded. During logs analysis, if there are inconsistencies in the behaviours between expected (GUI specifications) and actually tested (implementation), the execution logs will be further analysed by a replay-based debugger to find a reason of the behavioural inconsistencies.

Results The authors evaluated ADAutomation on two industrial mobile apps: KODAK PicFlick, and Newsyc using a case-study approach. The experimental results show that ADAutomation not only reduces the overall test time but also effectively detects critical faults in complex apps GUIs.

3.1.20 SwiftHand

Introduction In this work [12], the authors propose an automated technique for generating sequences of test inputs for Android apps, namely SwiftHand. The technique uses a machine learning algorithm to learn an approximate model of the target app. The learned model is used to systematically explore app states and refine itself during the app execution. A key feature of SwiftHand is to minimise app restarts by learning an approximate model of an app, which is usually sufficient to guide the testing procedure and visit unexplored app states.

Motivation The previous automated testing techniques usually learn a precise model of the mobile apps GUIs, which is a computationally intensive task. Due to the model precision, in order to explore specific app states which are reachable from the initial state, the existing automatic exploration techniques need to restart the app, which is a lengthy operation. The experiments show that the app restart operation takes around 30 secs, which is significantly longer than the time required to transit from one app state to another one. So, to address the problem with a massive number of app restarts, the authors propose a novel exploration strategy and learning algorithm which minimises the number of app restarts by learning an approximate model from the app.

Approach The SwiftHand design is guided by four practical assumptions: Testing Interface, Cost Model, User-Interface Model, and Test Coverage Criteria. Such design

criteria allow SwiftHand to reduce the number of app restarts by learning an approximate model, and achieve high code coverage quickly while systematically exploring app states. SwiftHand learns an approximate model of the GUI using the execution traces which are generated during the testing process. Afterwards, the learned model is used to select next user inputs which could potentially discover new app states. During execution, SwiftHand uses the newly discovered app state to expand the learned model, and refine it if there are any discrepancies between the model learned so far. Dynamic model learning and generation of user inputs helps SwiftHand to quickly explore states of the app, minimise the number of app restarts by searching for the execution paths leading to yet unexplored apps states using only user inputs.

Results The authors evaluated the performance of SwiftHand on several real-world Android apps and compared it with random and L*-based testing. The benchmark consists of 10 apps obtained from F-Droid [98]. The apps with native libraries (not written in Java), or requiring the Internet were excluded. For each app, the authors set three hours as a timeout for every test strategy. The experimental results show that, in a given time budget, SwiftHand achieves significantly higher coverage, and also reaches peak coverage faster than conventional random and L*-based testing.

3.1.21 Collider

Introduction In this work [136], the authors propose a two-phase technique for automated testing of the event-driven systems, namely Collider. It automatically generates event sequences which aim to reach a given target line in the app code. Collider specifically designed to generate event sequences for the targets which require long event sequences and reasoning about specific event parameters. In the first phase, Collider performs a concolic execution to identify individual event handlers of the target app. In the second phase, Collider forms event sequences reaching the target line, which is based on the identified event handlers from the first phase and the UI model of the target app.

Motivation A central challenge in automated mobile app testing is how to address the state explosion problem since possible event sequences are generally unlimited [8]. Current state-of-the-art black-box testing techniques such as random testing [10, 95], crawling-based [3, 13, 158], model-based [88, 159], static analysis [160] cannot effectively reach certain branches of the app code which are highly constrained by the event parameters. In contrast, symbolic execution can find required event parameters. However, most existing symbolic execution techniques [8, 161] cannot effectively construct long event sequences, and thus usually limited by the sequence length. So, to address the problem of reaching code fragments which require long event sequences with highly constrained

event parameters, the authors propose a targeted approach to automatically, yet efficiently generate such event sequences.

Approach The approach is inspired by [162] which addresses the problem of line reachability in C programs. Collider works in two phases. In the first phase, it performs concolic execution [75] of each event handler to collect symbolic path constraints for its execution paths, and prepares event handler summaries [163]. In the second phase, Collider uses the event handler summaries together with the given target app code line, and a UI model of the app to form a particular event sequence which leads to the target code line.

The generated event sequence is represented by an actual execution path which consists of the event handlers leading to the given target. Such an execution path (target path) is incrementally refined by searching for the event handlers which should be prior triggered to satisfy certain conditions in the target path. It uses the UI model and the symbolic states to identify and exclude the event handlers which are not relevant to the target path construction. When candidate event handlers are found, Collider forms the target path and verifies its satisfiability. This process continues until the entry state of the UI model is reached.

Results The authors evaluated the performance of Collider on five real-world Android apps. They specially selected apps with the targets which could not be easily reached by conventional random or model-based testing techniques. The experimental results show that Collider can efficiently produce event sequences for many highly constrained code targets.

3.1.22 A³E

Introduction In this work [11], the authors present a testing technique, Automatic Android App Explorer (A³E), to systematically explore Android apps while running on physical mobiles devices and not requiring app source code. The key idea of the technique is to use a static, taint-style, and data-flow analyses on the app bytecode in a novel way. Using static program analysis, A³E constructs a control-flow graph which captures transitions between app activities. Afterwards, this graph is used to guide an exploration strategy, namely Targeted Exploration. Targeted Exploration allows it to perform fast and direct exploration of activities. In addition, A³E implements Depth-First Exploration which simulates user actions to explore app activities. The Depth-First Exploration performs slower but systematically explores app activities.

Motivation Previous techniques for automated GUI exploration [8, 13, 45, 119, 164] have several limitations such as using small open-source apps, building incomplete app models, and an explosion of the state space. Furthermore, existing approaches have difficulties of adequate testing due to running the apps on emulators where various hardware components such as camera, GPS, microphone, flashlight and others are unavailable. In the light of these limitations, the authors develop A³E, a testing technique which systematically explores real-world Android apps running on physical mobile devices.

Approach A³E implements two testing techniques: Targeted Exploration and Depth-First Exploration. These strategies are not complementary and designed to achieve different goals. They both can start the exploration from the app entry points, mimic user actions on the app GUI, and trigger event handlers to invoke specific app activities.

Targeted Exploration is a novel technique which leverages static taint analysis to facilitate effective exploration of app activities. Targeted Exploration is designed to directly invoke special app activities since not all activities can be invoked through a user interface. Targeted Exploration aims to perform fast activity exploration which results in high activity coverage. To enable Targeted Exploration, A³E performs static bytecode analysis to extract a Static Activity Transition Graph which is further used to explore the graph systematically.

Depth-First Exploration is a fully dynamic testing technique which performs an automated GUI exploration in a depth-first manner. Depth-First Exploration aims to systematically explore app states which results in high method coverage. Depth-First Exploration mimics actual user actions on the app GUI, e.g., it clicks on UI widgets, or injects text into editable text boxes. Doing so, Depth-First Exploration transits to yet unexplored activities or returns to the previous one from where it continues the exploration procedure.

Results For the performance evaluation, the authors downloaded 25 real-world Android apps from Google Play. A³E focuses on improving coverage in two dimensions: activity, and method. So, to evaluate the effectiveness of A³E, the authors use two coverage metrics: activity coverage (number of screens explored), and method coverage (number of methods executed). The experimental results show that A³E is an effective technique. On average, it achieved 64.11% and 59.39% activity coverage using Targeted and Depth-First Exploration, respectively. It also achieves 29.53% and 36.46% method coverage using Targeted and Depth-First Exploration, respectively. A³E is also efficient. On average, it takes 74 secs for Static Activity Transition Graph construction, 87 mins for Targeted Exploration and 104 mins for Depth-First Exploration.

3.1.23 ORBIT

Introduction In this work [119], the authors present a novel grey-box approach for an automatic model extraction from a target mobile app. They present a tool ORBIT for the testing Android apps, which implements this approach. ORBIT uses static analysis to extract the events which are supported by the app GUI. Next, it performs dynamic crawling to systematically explore the app GUI by executing the extracted events on the target app.

Motivation Model-based testing is a popular, and important testing approach that operates on a model of an app’s behaviour. One of the important steps in model-based testing is the construction of the model itself. The model is typically an abstract representation of the app behaviour and may be constructed either manually [159] or using automatic techniques [22]. Manual model construction is usually a tedious and error-prone process. Automatic testing techniques [13, 22, 142] try to partially or completely automate the process of extracting models from GUI applications. However, such automatically constructed models are often of insufficient quality. As such, the authors develop an automated approach which efficiently generates high-quality models where their quality is evaluated by coverage, precision, and compactness criteria.

Approach The ORBIT consists of two components: Action Detector, and Dynamic Crawler. The Action Detector is implemented using the WALA, a static-analysis framework. Using static analysis, Action Detector extracts sub-call-graphs which are derived from the event-handler callback methods. Next, Action Detector analyses the app’s source code and manifest file to automatically build a map of intent-sending methods and intent filters by connecting the sub-call-graphs into a partially connected call-graph. The partial call-graph does not include broadcast intents whose behaviour is triggered at runtime. As such, such intents cannot be captured statically. Using the partially connected call-graph, the Action Detector executes an action-inference algorithm to generate the action map. Afterwards, the Dynamic Crawler executes a list of the actions from the map. Although the Dynamic Crawler selects actions from the action map, it is capable of handling dynamically-created and system-generated GUI components which are not statically declared.

Results To evaluate the performance of ORBIT, the authors compared it with Android GUITAR,¹ Android GUI Ripper [13], and Android Monkey [95]. The empirical evaluation shows that ORBIT can efficiently extract compact yet high-quality models in an automated manner. The execution time and coverage results show that ORBIT 32%–75%

¹not available, as of March, 2018

faster while constructing 5%–140% more accurate models than those which are built by Android GUITAR, and Android GUI Ripper.

3.1.24 Dynodroid

Introduction In this work [10], the authors present a system, namely Dynodroid, for generating relevant inputs to Android apps. Dynodroid interacts with the target app via a sequence of events which are generated using the Android framework. Dynodroid instruments the Android framework so that it can observe app behaviour and use it to guide the generation of the next event to the app. Dynodroid also allows interleaving automatically generated events with events generated by humans.

Motivation Program analyses are broadly classified into static and dynamic. Static analyses are usually hindered by app code obfuscation, using native libraries, and a complex SDK framework. As a result, there is growing interest in dynamic analyses of mobile apps (e.g., [165–167]). A key challenge here is to generate relevant program inputs in an automated manner, which can adequately exercise the program’s functionality. As such, the authors are interested to build a system for generating relevant inputs to mobile apps on Android. They identify five key criteria which system must satisfy: robust, black-box, versatile, automated, and efficient.

Approach The main Dynodroid principle is based on an “observe-select-execute” cycle. First, it observes events which are relevant to the current app state. Next, it selects one of those events and then executes the selected event to produce a new state in which it repeats this process. For UI events, the relevant events are identified via the program’s user interface, e.g., tap or a gesture on the device’s screen. For system events, Dynodroid identifies a relevant to the app event by checking which one has registered a listener for such event, e.g., incoming SMS message. For textual input generation, Dynodroid allows a user to interact with the target app. The user can pause the testing process, and manually generate required textual inputs, or complex touch events.

Results The authors evaluated Dynodroid on 50 open-source Android apps. They compared it with two prevalent approaches: manual testing and automated fuzzing (Monkey) [95]. Dynodroid, humans, and Monkey covered 55%, 60%, and 53%, respectively, of each app’s Java source code on average. Monkey took 20X more events on average than Dynodroid. Dynodroid also hit nine bugs in seven of the 50 apps, and six bugs in five of the top 1,000 free apps downloaded from Google Play.

3.1.25 ACTEve

Introduction In this work [8], the authors present an automated concolic testing technique for event-driven systems, namely ACTEve. ACTEve systematically and automatically generates sequences of the relevant input events for Android apps. It aims to achieve high structural code coverage, while eliminating the path-explosion problem [163]. The path-explosion is a common problem to symbolic and concolic testing [56, 163, 168–174]. ACTEve mitigates it by checking a condition on the execution paths, which identifies subsumption between different event sequences.

Motivation Static analysis of mobile apps is innately a challenging task. The app code and architectural complexity may depend on Android SDK, asynchrony, inter-process communication, databases, obfuscation, and GUIs. As such, many automated approaches tend to implement dynamic analysis (e.g., [165, 175, 176]) techniques. However, the performance of a dynamic analysis highly depends on the relevant program inputs (e.g., user gestures, or textual) which should be generated during execution. So, to provide sensible program inputs, existing approaches commonly use either (1) capture-replay techniques to automatically replay the previously manually crafted model of the app GUI [22, 159] with different program inputs, or (2) model-based techniques which usually require users to provide an accurate model [45, 177] since automatic model extraction is usually incomplete and of poor quality. However, these techniques have their limitations, e.g., capture-replay is usually highly dependent on a particular platform event-handling mechanism, and model-based approaches usually require tremendous manual effort for the model construction. So, to address this problem, the authors propose a new approach, which is both general and automatic.

Approach ACTEve implements a systematic test-input generation technique which is based on concolic testing [75, 76, 178] (also known as dynamic symbolic execution). It symbolically tracks events from GUI where they originate to the code where they are handled (event handlers). The underlying idea of ACTEve is to identify subsumption between two event sequences so that the event sequences can be conditionally pruned without scarifying test coverage. ACTEve computes subsumption between event sequences by checking for simple data- and control-flow information of each execution path individually.

ACTEve identifies itself by four fundamental properties. First, ACTEve is built for real-world mobile apps, so it does not work on any unrealistic assumptions about the target app. Second, ACTEve has a small memory footprint, so it does not store any program traces or execution states. Third, ACTEve is a lightweight technique, so it does not require heavy constraint solvers since every app is analysed individually. Fourth,

ACTEve does not perform analysis of the app code which is beyond the scope of static analysis.

Results The authors show the performance of our ACTEve on five Android apps. They compare ACTEve results with exhaustive (conventional) concolic testing. The experimental results show that ACTEve is significantly more efficient than exhaustive testing. ACTEve requires 5%–36% of the running time of exhaustive execution to achieve the same structural coverage. ACTEve makes it feasible by pruning multiple execution paths which do not expose new app behaviours. For example, for the Music Player Android app, using a maximal event sequence length, i.e., ($k = 4$), ACTEve explores only 16% of the possible inputs (3,445 of 21,117) while achieving the same structural coverage as the exhaustive testing.

3.2 Discussion

In this section, we discuss effectiveness, and efficiency of the described automated GUI testing techniques for mobile apps.

3.2.1 Effectiveness

Effectiveness is one of the important characteristics of automated testing techniques. For the automated testing techniques, effectiveness is usually assessed through the coverage which can be achieved upon completion of tests execution. In particular, code coverage metric is the useful means of the effectiveness assessment for the automated testing techniques.

We identify four conceptual characteristics of the automated testing tools which may affect effectiveness: *model paradigm*,² *test generation approach*,³ *test generation criteria*,⁴ and *textual input generation*.⁵ From our description of the 25 described techniques, we conclude that test generation approach plays a dominant role in effectiveness. It innately identifies model paradigm and test generation criteria, which subsequently impact the effectiveness of the testing technique. There is also one additional characteristic such as a textual input generation which affects the effectiveness. Depending on the model

²It describes nature of the built model, i.e., which modelling notations are used to describe the behaviour of the target apps for test generation purposes.

³It describes how tests are derived from a built model. It may also combine several approaches to facilitate the non-trivial task of automated test generation from a model.

⁴It defines test criteria which are used to control the generation of tests. These criteria indirectly define properties of the generated test suites, including their fault detection capability, cardinality, and structural complexity.

⁵It describes what kinds of textual user-inputs can be generated by the artefact. The input generation process itself can be automated or manual, while the input kinds can be random, concrete, predefined, contextual, or others.

paradigm, it may impact the effectiveness to a different extent. For example, random-based and stochastic models are less likely to be affected due to their random nature, while deterministic models such as transition- or control-data-based could be affected severely. Also, we conclude that testing techniques which implement model-checking, symbolic execution, constraint solving, and search-based test generation approach tend to be more effective than those implementing random test generation. It could be explained by the fact that more sophisticated test generation approaches are likely to be more effective since they usually exploit the built model heuristics to generate high coverage tests.

We observe that random test generation approaches tend to use fault detection as a test generation criteria, while the others mainly focus on structural model or code coverage. It could be explained by the random nature of the testing approaches. Randomly generated tests are likely to expose more bugs due to their unexpected (random) nature, while systematic ones may not hit the same number of bugs unless their coverage is 100%. This is the main reason why systematic testing approaches focus on increasing structural model or code coverage aiming 100%. Otherwise, random techniques will always take the first place being the state-of-the-art and practice in automated functional testing.

We also observe that more sophisticated textual input generation mechanisms help to improve the effectiveness of the automated testing techniques. It could be explained by the fact that mobile apps have highly interactive nature, and thus they are crafted with a mindset of being used by humans, not machines. As such, various app behaviours highly depend on user inputs, textual in particular. So, currently, such app behaviours cannot be adequately tested by automated techniques due to the lack of adequacy in the automatic textual input generation.

3.2.2 Efficiency

Efficiency is another important characteristic of automated testing techniques. For the automated testing techniques, efficiency is usually assessed through an execution time requiring the testing process to complete. However, the efficiency of the automated testing techniques may vary depending on various factors such as experimental data sets, execution environment, computation power used for the experiment.

We identify four conceptual characteristics of the automated testing tools which may affect efficiency: *testing approach*,⁶ *testing technique*,⁷ *search algorithm*,⁸ and

⁶It describes which kind of the model is used for the automated testing. We identify two main model kinds: one is derived from “GUI” (user interface/event flow), and the other one is derived from “Code” (source or binary) of the target app.

⁷It describes in which manner an artefact performs an exploration of the built model. We identify two main approaches: one is “Systematic”, where the technique implements a guided exploration, and the other one is “Random”, where the technique implements a random-based exploration strategy of the built model.

⁸It describes which algorithm is used, or based on to guide the exploration process. We identify two main algorithms: one is “Guided” which uses various model heuristics to guide the search, and the other one is “Random” which is based on uniform random and probabilistic events generation.

termination condition.⁹ From our description of the 25 described techniques, we conclude that testing approach plays a dominant role in efficiency. Another characteristic such as search algorithm innately identifies testing technique which may subsequently impact efficiency to a different extent. For example, sophisticated search algorithms may require more complex implementation which eventually may slow down the overall efficiency of the testing technique, while simple techniques, such as random, are unlikely to impact the efficiency due to their simplicity. There is also one additional characteristic such as a termination condition which affects the efficiency. The termination condition is innately identified by the testing technique. Depending on the testing technique, it can be automatically derived from the conditions of the constructed model during runtime. However, it can also be specified manually by a user to indicate when the testing procedure shall stop. For example, systematic testing techniques tend to use automatic termination condition, while random ones usually rely on the user-specified termination conditions. Also, we conclude that testing techniques which implement code search-based testing approaches tend to be more efficient than those implementing GUI model-based. It could be explained by the fact that code search-based testing approaches generate tests being guided by more simple and compact models inferred from the app code rather than app GUI. The complexity of the GUI models could be much higher, and their model size could be much larger than those derived from the app code.

We observe that systematic testing techniques tend to be less efficient than random ones. It could be explained by the fact that systematic techniques require more advanced search algorithms to implement a guided search, while random ones usually rely on uniform random. In contrast, systematic techniques use various heuristics of the inferred model to guide the testing process, while random ones do not require any heuristics so that they drive the testing process by sampling random actions from the uniform distribution.

We also observe that termination condition which relies on a user-specified condition is less efficient than the one which is automatically derived from the model properties. It could be explained by the fact that if the user specifies a termination condition, it usually implies that the testing approach is not capable of constructing a high quality or deterministic model so that the termination condition cannot be derived from the model properties during runtime. However, if the termination condition is automatically derived from the model properties, it likely implies that the constructed models are of high quality and deterministic enough so that the model properties can be used to automatically determine when to stop the testing process.

⁹It describes a condition which is to be satisfied to terminate the testing process. The termination condition can be determined either by the model properties or manually by the user.

3.3 Conclusion

From effectiveness, we observe that the existing automated testing techniques are not effective enough, and currently, they achieve nearly half of the desired level of effectiveness. As such, there is still a gap which requires further research to improve existing techniques or develop conceptually new test generation approaches to improve effectiveness. In addition, we highlight another area for improvement such as automatic textual input generation. Our observations show that most of the techniques currently use random text generation which may significantly impact the desired effectiveness, especially for mobile apps. However, automatic relevant input generation is not a trivial task; this is why it is still an evolving field.

From efficiency, we observe that current automated testing techniques are not efficient enough. In general, they provide medium-to-low efficiency requiring more than 30 minutes per app. Specific techniques even set several hours per app to more or less adequately explore app functionality. As such, there is still a significant gap which requires further research to improve the efficiency of the automated testing techniques. In addition, we highlight another area for improvement focusing on the construction of compact yet high-quality models, thus reducing their size and complexity. Existing approaches usually infer the models “as are”, which commonly results in large and complex models. Even their quality is usually high; however, exploration time may grow exponentially to complete traversing such large models. The exploration time may take very long since existing search algorithms are not well-adapted to perform on such large models where exploration state-space could be infinite.

Chapter 4

Automated Functional Testing of Mobile Apps GUI

4.1 Summary

Automated functional user interface (UI) testing of mobile apps is a challenging task due to the highly interactive nature of app UIs, and thus it commonly fails to provide high code coverage. In this chapter, we present DroidDEV, an automated UI exerciser which aims to achieve high code coverage for the real-world Android apps. DroidDEV dynamically builds a finite UI flow graph, generates and injects UI-context-aware inputs to exercise all widgets on all discovered UI screens through a best-first search pathfinding algorithm. We have evaluated DroidDEV on 20 real-world open-source Android apps and compared it with manual UI testing and two other prevalent automated approaches including MobiGUITAR and Monkey. On these subjects, DroidDEV reports code coverage similar to manual UI testing, and it outperforms MobiGUITAR and Monkey regarding code coverage and exercising time.

4.2 Introduction

The Android apps are becoming increasingly prevalent, yet due to the pressing time to market, numerous apps are being released without sufficient testing and auditing. Thus, there is a growing need for automated software quality assurance tools. A central challenge in automated testing is a generation of tests with high coverage due to the highly interactive nature of the mobile apps' graphical user interfaces (GUIs) [1, 3]. In addition, existing GUI testing techniques [4–7] for automated testing of PC-based applications are no longer feasible to perform GUI testing of the mobile apps due to their specific non-trivial GUI structure [2, 3]. Consequently, in recent years, many automated GUI testing systems for the mobile apps have been developed [2, 8, 10–14]. However, in

practice, they are still incapable of achieving high code coverage [14, 15] because they fail to simultaneously address both of the following major problems:

- The mobile apps have a specific non-trivial GUI structure. As such, a large number of execution paths should be explored via the non-trivial apps GUIs. Therefore, it has become impractical to automatically achieve high code coverage within a reasonable time.
- The mobile apps GUIs are innately highly interactive because they commonly rely on sensible user inputs which are provided by humans. However, in practice, generating non-trivial user inputs in an automated manner is not an easy task.

To overcome limitations of existing UI testing approaches, in this chapter, we propose a novel approach, namely DroidDEV (AnDroid Dynamic, Exerciser, high coVerage). It achieves high code coverage in the mobile apps by exhaustively and systematically discovering app UIs at any depth in the GUI model and automatically generating UI-context-aware inputs. DroidDEV performs functional automated GUI testing via an implementation of a black-box testing approach. It builds concrete app GUI models and uses them for guiding a testing process, and generates relevant user inputs for automated app exercising. As such, DroidDEV innately belongs to a class of the testing tools which perform an automated GUI exploration via active learning. So, since DroidDEV performs an automated app exercising via the GUI exploration with active learning, it does not use any abstract app GUI models. Also, since DroidDEV is a testing technique, it does not intend to perform as a software modelling [179, 180] or software development [181, 182] framework. As such, DroidDEV does not facilitate the mobile apps' developers to examine various app GUI design alternatives, or evaluate many diverse configuration possibilities, or automatically generate source code, and others. Instead, DroidDEV aims to automate an app GUI testing process by minimising or eliminating manual efforts.

DroidDEV automatically guides a GUI testing procedure by dynamically building a finite state model without loops of a target app GUI [101]. DroidDEV is built by extending the *Google's* UI Automator testing framework.¹ The UI Automator testing framework is designed for writing black box-style functional automated UI tests, where its test code does not rely on internal implementation details of a target app. Other than the UI Automator, there are other automated UI testing frameworks such as MonkeyRunner,² Troyd,³ Robotium,⁴ Appium,⁵ and Android GUITAR.⁶ Unlike these, the UI Automator provides a unique feature to dynamically dump a view hierarchy of the foreground app

¹<https://developer.android.com/training/testing/ui-automator>

²<https://developer.android.com/studio/test/monkeyrunner>

³<https://github.com/plum-umd/troyd>

⁴<http://code.google.com/p/robotium>

⁵<http://appium.io>

⁶<http://sourceforge.net/projects/guitar>

UI displaying on an Android device screen into XML. From the XML data, DroidDEV builds a finite state model without loops of a target app GUI, and thus guides its testing procedure.

DroidDEV explores an app GUI using our novel f -UFG model (see Sections 4.3.2, and 4.3.3). It dynamically builds a finite UI flow graph (f -UFG) of an app GUI model, and systematically traverses it by implementing an informed search algorithm A* via a best-first search procedure, which is the best-known form of the *Best*-first search (BFS) algorithm [102]. The A* search algorithm combines BFS for efficiency with the uniform cost search for optimality and completeness. The key idea behind the A* search algorithm is to efficiently generate the shortest sequences of the UI events leading to a target app UI.

In this chapter, we have made the following contributions:

1. We propose a novel approach, namely DroidDEV, which provides high code coverage in automated UI testing of the Android apps through dynamically refining a finite UI flow graph of the apps, and an automated UI-context-aware input generation.
2. We implement DroidDEV for the Android apps, taking into account their specific non-trivial structure of GUIs.
3. We conduct an experiment on 20 real-world open-source Android apps to evaluate the performance of our approach regarding code coverage and exercising time with manual UI testing and two prevalent automated approaches including Monkey and MobiGUITAR.

4.3 Proposed Approach: DroidDEV

In this section, we provide detailed explanations of the proposed approach DroidDEV, focusing on the algorithms it employs to perform fully automated exercising of the Android apps UI to achieve high code coverage. In Section 4.3.1, we state the main problem which DroidDEV addresses. In Section 4.3.2, we give several conceptual definitions for DroidDEV explanations. In Section 4.3.4, we provide an overview of the proposed approach. In Section 4.3.5, we describe three main algorithms used by DroidDEV including SensINJECT, DynaEXE and BestPath for the automated input generation, dynamic execution and best-first search pathfinding algorithms, respectively.

4.3.1 Problem statement

Majority of the mobile apps have a non-trivial GUI structure, i.e., their UI models are not finite state and/or containing loops. Therefore, it is a challenge to perform

an automated GUI testing on these apps. For example, existing tools for the GUI testing of the mobile apps build a GUI model, and use it “as is” for a systematic exploration [2, 10–13, 15, 88, 103, 119, 136]. However, in fact, the GUI models of the mobile apps may innately have an infinite number of UIs, or may have infinite loops. As such, the GUI-testing procedure may infinitely exercise an app unless a user [16] explicitly specifies a termination condition. So, to address this problem, existing tools set a termination condition for a testing procedure such as the limited execution time, or the manually predefined number of events to be injected, or the depth of exploration in the GUI model [15, 16]. Therefore, with such specified termination conditions, existing tools may not discover certain app UIs, especially if they are located at deep levels in the GUI model.

4.3.2 Definitions

For further explanations of DroidDEV, in this section, we provide three definitions which are used throughout this chapter.

Definition 1: User Interface (UI) is a view hierarchy of the objects (i.e., widgets, or controls), which define a layout of a single app screen; The objects may be input widgets (editable), and other widgets (non-editable). This definition states that each app UI usually has one or more UI widgets which could be editable (e.g., “*EditText*”) or non-editable (e.g., “*Button*”). A non-editable UI widget allows a transition to the same or another app UI.

Within the context of this chapter, we assume that two UIs are equivalent if (1) the number of UI widgets is equal, and (2) their values of the attribute “*class*” match when comparing them from the top to the bottom for two UIs (i.e., view hierarchies). Note that the UI widget attributes “*text*” (only for editable widgets), “*NAF*”, “*bounds*”, “*focusable*”, “*enabled*”, “*checkable*”, “*scrollable*”, “*long-clickable*”, “*selected*”, “*focused*”, “*clickable*”, “*checked*” are excluded from the comparison of two UIs since these attributes do not imply an equivalence of two UIs.

Definition 2: UI Flow Graph (UFG) is defined as a triple $g = (U, W, L)$, where U is the set of non-equivalent UIs, W is the set of widgets on a UI, and L is the set of links, where each link l_i is a directed edge from a widget w_{ij} to UI u_q , denoted as $l_i = w_{ij} \rightarrow u_q$, where $l_i \in L$, $w_{ij} \in W$, $u_q \in U$ and w_{ij} is on u_q , where “ i ” is the index of UI in UFG, and “ j ” is the index of the widget on UI, where $i > 0$, $j > 0$, $0 < q \leq i + 1$.

An example of *UFG* is shown in Figure 4.1. In Figure 4.1, u_i represents a UI, e_{ij} represents an editable widget, and w_{ij} represents a non-editable widget, where “ i ” is a sequential index of the UI, and “ j ” is a sequential index of the widget on the UI. On each UI, w_{ij} allows a transition to the same or another UI as indicated by the arrows. In Figure 4.1, the *UFG* is a finite state model, however it has several loops inside. For

example, the following sequence of links $l_1 = (w_{12}, u_2), l_2 = (w_{21}, u_3), l_3 = (w_{32}, u_4), l_4 = (w_{41}, u_1)$ forms a *loop* in *UFG*.

Definition 3: Finite State UFG (*f-UFG*) is an *UFG* which is finite state and does not contain any sequence of links $l_i, l_{i+1}, \dots, l_n$ which forms a loop from $u_i, u_{i+1}, \dots, u_m$, where $l_n = w_{nj} \rightarrow u_m, l_n \in L, u_m \in U$, and $i > 0, j > 0, n > 0, m > 0, i \leq n, i \leq m, m \leq n$. This definition states that *f-UFG* is a finite state *UFG* model which does not contain any sequence of links forming a “loop” between any two UIs in the *UFG* model.

Figure 4.1: Example UFG of mobile app

4.3.3 Novel aspects

DroidDEV distinguishes itself from existing techniques by the following novel features:

1. In DroidDEV, we introduce a novel feature such as the *f-UFG* model of an app GUI. The *f-UFG* model is a finite state GUI model of the mobile app, which does not have any loops at any point in time. The *f-UFG* model enables DroidDEV to have the following improvements:
 - (a) It enables DroidDEV to notably reduce the exercising time. Instead of using a standard implementation of *Depth-first*, or *Breadth-first* search or Random GUI exploration algorithms, DroidDEV implements an ad-hoc GUI exploration strategy which is based on the existing A* technique (see Section 4.4.5). It allows DroidDEV, during the exercising process, to quickly find the shortest path to the lastly discovered app UI from which the exercising process continues. As such, an overall exercising time is reduced which makes DroidDEV

more efficient in comparison with other automated approaches discussed in this chapter.

- (b) It enables DroidDEV to notably increase the efficacy of an automated GUI exploration. Without falling into an infinite loop, DroidDEV discovers as many UIs existing in an app GUI model as possible. As such, DroidDEV demonstrates significantly higher code coverage results in comparison with

Algorithm 1: DroidDEV

Input: *app* - application under test (AUT)

```

1 procedure DroidDEV(app)
2   coveredUIs =  $\emptyset$ ;
3   f-UFG =  $\emptyset$ ;
4   app.Run();
5   currentUI = app.GetCurrentUI();
6   while (TRUE) do
7     if (currentUI  $\notin$  f-UFG) then
8       | f-UFG.AddUI(currentUI);
9     end
10    foreach (editable widget eWid in currentUI) do
11      | SensINJECT(eWid);
12    end
13    nWid = GetUnExercisedRelevantWidget(currentUI);
14    continuity = DynaEXE(currentUI, nWid);
15    lastUI = f-UFG.GetLastlyDiscoveredUI();
16    if (lastUI.isFullyExercised()) then
17      | coveredUIs.AddUI(lastUI);
18      | f-UFG.Remove(lastUI);
19    end
20    currentUI = app.GetCurrentUI();
21    if (currentUI  $\notin$  f-UFG and currentUI  $\in$  coveredUIs) then
22      | PressBack(app);
23      | currentUI = app.GetCurrentUI();
24      | if (currentUI  $\notin$  f-UFG) then
25        | | continue while;
26      | end
27    end
28    if (continuity == FALSE) then
29      | p = BestPath(currentUI, f-UFG);
30      | currentUI = Retrace(p);
31    end
32    if (f-UFG ==  $\emptyset$  or f-UFG  $\in$  coveredUIs) then
33      | break while;
34    end
35  end
36 end

```

other automated approaches discussed in this chapter.

2. In DroidDEV, we introduce a novel feature such as ***UI-context-aware input generation*** mechanism.

- (a) It enables DroidDEV to generate UI-context-aware user inputs. DroidDEV uses a manually crafted dictionary of basic text terms (DBT2) which is, in turn, built on the textual terms which are extracted from multiple reputable apps. The idea of the UI-context-aware inputs is to automatically, during the exercising process, obtain relevant input values which lie in a set of possible values with a specific input pattern, e.g., email address, or server IP address. So, it allows DroidDEV to automatically resolve such situations where the user inputs with a specific pattern are required, and thus DroidDEV avoids human interaction in the testing process.

4.3.4 Overview

An overall workflow of DroidDEV is given in Algorithm 1. Initially, a set of covered UIs (*coveredUIs*, line 2) and *f-UFG* (line 3) are set to empty. DroidDEV runs a target app (line 4) to render the first app UI (line 5). Next, the *f-UFG* model of the app is dynamically built and refined (lines 6–35) as long as DroidDEV exercises the app, i.e., until all the UI widgets on all the discovered app UIs have been exercised. At every iteration of the exercising cycle, DroidDEV adds the current app UI with its widgets into the *f-UFG* model if it is not in the model (lines 7–9). The UI-context-aware inputs will be automatically injected into all the editable UI widgets using the DroidDEV’s SensINJECT algorithm (lines 10–12). Afterwards, each non-editable unexercised relevant UI widget *nWid* of the current app UI will be retrieved (line 13) and automatically exercised by DroidDEV’s dynamic execution engine, namely DynaEXE (line 14). DynaEXE works by generating the relevant UI-events to exercise the non-editable widget *nWid*, depending on a type of the UI widget. For a comprehensive app exercising, DynaEXE supports the most common types of the UI-events. The set of UI-events that are currently supported by DroidDEV is given in Table 4.1.

Exercising the *nWid* renders a subsequent UI screen. Afterwards, DynaEXE adds the link from the *nWid* to the subsequent UI screen in the *f-UFG* model if the subsequent UI screen is not in the model. As such, DynaEXE ensures that the *f-UFG* model does not have any loops. During execution, DynaEXE monitors the execution flow. If there is any disruption in the execution flow, i.e., a widget transition that leads to a loop in the *f-UFG* model, DynaEXE returns FALSE as a result of the execution to indicate a discontinuity in the execution flow (line 14), otherwise returns TRUE. After DynaEXE execution, DroidDEV obtains the lastly discovered app UI from the *f-UFG* model (line 15). The

lastly discovered UI is a UI that was last added into the *f-UFG* model. DroidDEV checks if the lastly discovered UI is fully exercised (i.e., all its widgets are exercised) (line 16). If so, DroidDEV adds the lastly discovered UI into the covered UIs (line 17) and removes it from the *f-UFG* model (line 18). Afterwards, the lastly discovered app UI in the *f-UFG* model is changed.

In line 20, DroidDEV updates the current UI after DynaEXE execution. If the current UI on a testing device is not in the *f-UFG* model but has been fully exercised (line 21), DroidDEV will not proceed with the current UI. Instead, DroidDEV generates a system key event “Back” (line 22). This event returns to any previously discovered app UI. Thus, the current UI is updated (line 23). If the current UI is not in the *f-UFG* model (line 24), DroidDEV continues the exercising procedure (line 25). Note that, depending on app logic, system key event “Back” may exit the app. Such the case is handled by DroidDEV by restarting the app and continuing the exercising procedure from the lastly discovered UI in the *f-UFG* model. If DynaEXE does not indicate continuity in the execution flow (line 28), DroidDEV finds a path to the lastly discovered UI in the *f-UFG* model (line 29) and retraces the found path to reach the lastly discovered UI (line 30). A retracing procedure (line 30) goes through all the widgets on the path p , generating inputs using SensINJECT for the editable UI widgets and exercising non-editable UI widgets along the path to reach the lastly discovered UI in the *f-UFG* model. This is an important part of DroidDEV which ensures that if there is any discontinuity in the exercising procedure, DroidDEV can track back the lastly discovered UI and exercise its yet unexercised UI widgets. DroidDEV automatically terminates the exercising process once the *f-UFG* model is empty or a set of the covered UIs contains a set of the UIs from the *f-UFG* model (lines 32–34).

4.3.5 Components

DroidDEV employs the following three main components:

1. **SensINJECT (Sensible UI-context-aware input INJECTION)** automatically injects UI-context-aware inputs to a foreground app UI which is currently displaying on the Android device screen.

Table 4.1: Types of events supported by DroidDEV

Event type	Action(s)	Hardware key(s)
Touch	Click; LongClick	–
Motion	Swipe; Scroll	–
Trackball	Roll; Press	–
Keypress	Input Injection	–
“Major” navigation	–	“Menu”
“System”	–	“Home”; “Back”

2. **DynaEXE (Dynamic EXEerciser)** automatically exercises a non-editable UI widget by generating a relevant UI-event at a time. DynaEXE ensures that the f -UFG model does not have any loops.
3. **BestPath (Best-first search Path finding algorithm)** performs a UI search in order to automatically build a path leading to the lastly discovered UI which was last added into the f -UFG model.

In the following sections, we explain each of their implemented techniques in detail.

SensINJECT: UI-context-aware textual input generation

DroidDEV automatically injects UI context-aware inputs to editable UI widgets by performing a context analysis of a given editable UI widget. In Algorithm 2, we provide details of the input injection process for an editable UI widget ($eWid$), namely SensINJECT. SensINJECT extracts relevant input types from three UI widget attributes including “*text*”, “*resource-id*” and “*content-desc*”. SensINJECT identifies a relevant input type of the editable UI widget (line 2) by referring to a dictionary called SensDIC, which essentially consists of the commonly used input types and their corresponding input values. We describe SensDIC in more details in the following sub-sections. Based on an input type, SensINJECT searches for the input value in SensDIC (line 4) and injects it into the relevant editable widget (line 5). As SensDIC does not contain all possible input types, there might be a chance where SensINJECT could not extract any input type from the widget attributes. In this case, SensINJECT applies an input generation procedure (lines 6–17) as follows:

- If an editable accessible widget contains a default value (i.e., actual value or hint) which is set by the app itself, SensINJECT saves this default value for a future input generation (line 9) and injects the value into the widget (line 10).
- If an editable accessible widget does not contain any default value, SensINJECT generates a random input as follows: (1) SensINJECT tries to inject a random numeric integer value within the [0, 100] range (line 12); (2) otherwise, if the numeric value is rejected by the app (e.g., only string values are allowed), SensINJECT injects a random string value (line 14).

How SensDIC was built? We build SensDIC as one part of the proposed approach for the input injection. SensDIC consists of commonly appeared editable widget input types such as *email address*, *password*, *login*, *server address*, *server port*, *URL*, and *phone*.

To build SensDIC, we manually inspected 100 selected apps from *Google Play*, which require complex input, such as Skype, Facebook and Email, and others. From the selected

Algorithm 2: SensINJECT

Input: *eWid* - editable UI widget

```
1 procedure SensINJECT(eWid)
2   inputType = SensDIC.GetInputType(eWid);
3   if (inputType  $\neq$   $\emptyset$ ) then
4     | inputValue = SensDIC.GetInputValue(inputType);
5     | Inject(inputValue, eWid);
6   else
7     | defaultValue = eWid.GetDefaultValue();
8     | if (defaultValue  $\neq$   $\emptyset$ ) then
9       | Save(defaultValue, eWid);
10      | Inject(defaultValue, eWid);
11    | else
12      | result = InjectRandomNumericValue(eWid);
13      | if (result == FALSE) then
14        | InjectRandomStringValue(eWid);
15      | end
16    | end
17  end
18 end
```

apps, we learn input types which are common for these 100 apps. So, we build SensDIC to identify input types which are required by an app under test during runtime. Based on the identified input type, SensDIC returns its relevant input value. Using XML dumps produced by the UI Automator testing framework for these 100 apps, we selected all the editable accessible UI widgets, and extracted the text terms of their specific UI attributes such as “*text*”, “*resource-id*”, and “*content-desc*”. We exploit the extracted text terms of these attributes to identify the most common input types such as *email address*, *password*, *login*, *server address*, *server port*, *URL*, and *phone*. These text terms usually reflect well an actual meaning of the required input types, although the attributes descriptions may vary depending on the developers. As such, the failure in identifying a suitable input type solely depends on the descriptions of the attributes. Knowing this fact, with a good chance of having adequate attributes descriptions, we selected 100 apps from trusted developers. As such, we believe that the identified text terms of the attributes are sufficient for the majority of the mobile apps due to their behavioural commonalities. Particularly in our experiment, the identified text terms were sufficient for SencDIC not to let it fail unless an editable widget does not require a specific input type, and can accept any random value (number or string). If a suitable input type is found in SencDIC, SensINJECT will be using the same value which corresponds to the identified input type until the testing process is complete. However, if the input type is not found in SencDIC, SensINJECT will generate random values which are different at every time of the input injection.

It is worth emphasising that SensDIC consists of input types with their corresponding input values. For example, for an editable UI widget which requires the *email* input type, SensDIC defines a default input value '*email@example.com*'. SensDIC contains default valid input values for known input types (see above), and thus, it does not require manual inputs configuration for every new app before its testing. Optionally, for the input types in SensDIC, the realistic textual inputs can be provided by a user in the project *config.properties* file so that they will be injected during app exercising instead of the default input values in SensDIC. In contrast, the existing automated tools, by default, tend to generate input values randomly unless their values are manually configured prior an app testing, or inputs can be injected manually during an app testing. However, manual inputs configuration will be required for every new app, which is a tedious procedure, while the manual inputs injection during an app runtime is error-prone.

Why SensDIC was built? SensDIC was built to overcome the limitations of state-of-the-art automated input generation approaches [10,95] where textual inputs are randomly generated, and they are trivial for most of the Android app UIs which are specifically designed for the human interaction. For example, if an app requires *login* and *password*, it is hard for an automated tool to generate reasonable values for these kinds of inputs, which are usually provided by a human user. As SensDIC contains commonly used input types, SensINJECT is context-aware of the input type of a particular editable UI widget and thus, can search and inject a suitable input value into the UI widget if the input type of the widget is in SensDIC.

DynaEXE: Automatic GUI exerciser

In Algorithm 3, we show the procedure for DynaEXE. An input for DynaEXE is the current UI (*currentUI*) and the non-editable widget (*nWid*) on the UI. DynaEXE uses the XML files produces by the UI Automator framework to understand the structure of the current UI under exercising. DynaEXE first inspects the attributes of the *nWid* and generates a suitable UI event for the UI widget, and executes the event for the widget (line 2). By firing a UI event, the current UI might be changed. As such, the current UI should be updated (line 3). If the current UI is a newly discovered UI screen which is not in the *f-UFG* model (line 4), a link from *nWid* to the current UI is added to the *f-UFG* model (line 5). DynaEXE returns TRUE (line 6) to indicate that there is no disruption in the execution flow and DroidDEV can continue with the newly discovered UI. Otherwise, DynaEXE returns FALSE (line 8) to indicate that there is a disruption in the execution flow, which requires DroidDEV to track back the lastly discovered UI in the *f-UFG* model.

Algorithm 3: DynaEXE

Input: *currentUI* - app UI; *nWid* - non-editable UI widget

Output: *continuity*

```
1 procedure DynaEXE(currentUI, nWid)
2   GenerateRelevantUIEvent(nWid);
3   currentUI = app.GetCurrentUI();
4   if (currentUI  $\notin$  f-UFG) then
5     | f-UFG.AddLink(nWid, currentUI);
6     | return continuity = TRUE;
7   end
8   return continuity = FALSE;
9 end
```

BestPath: Best-first search path finding

BestPath algorithm implements the Best-first search algorithm [102] guided by a heuristic to find a path to the lastly discovered UI in *f-UFG*. Our implementation of BestPath ensures that for the current *f-UFG* model (since it dynamically changes during an app GUI exploration), it always returns the shortest (best) path to the lastly discovered app UI. This is because the path to the lastly discovered app UI is determined by the visited UI widgets leading to this lastly discovered app UI (the *f-UFG* model keeps track of such UI widgets, and once a loop has been detected, it replays the best path to reach that app UI which was left off right before the loop has occurred). As such, there are no such circumstances under which the best path would not be found. Note again that the best path is only relevant to the current *f-UFG* model. Since the *f-UFG* model refines itself during an exercising procedure, the best path for each refined *f-UFG* model could be different. However, it still will be the shortest path in that particular *f-UFG* model.

BestPath is shown in Algorithm 4. BestPath starts from the current UI. A heuristic that is used to guide the search is as follow. On each UI, BestPath selects only one non-editable widget *rWid* that was most recently exercised and still has links (line 2).

Algorithm 4: BestPath

Input: *UI* - app UI, *f-UFG* - finite UFG

Output: path *p* to lastly discovered UI

```
1 procedure BestPath(UI, f-UFG)
2   rWid = GetRecentlyExercisedWidget(UI);
3   p.Add(UI, rWid);
4   foreach (outgoing link l from rWid) do
5     | p = BestPath(l.GetEndUI(), f-UFG);
6   end
7   return p;
8 end
```

The idea behind is that the chosen widget which satisfies the heuristic is likely to lead to the lastly discovered UI in the f -UFG model. The UI (UI) and the chosen widget $rWid$ are added to the path (line 3). For each outgoing link l from the selected widget $rWid$, BestPath recursively continues the search (lines 4–6). As each link l is a directed edge from a non-editable widget to a UI, GetEndUI function returns a subsequent UI where $rWid$ connects through the link. Note that the lastly discovered app UI does not have any outgoing links. Thus, when BestPath reaches the lastly discovered UI, it skips the loop (lines 4–6), and terminates by returning the best path p (line 7).

It is important to note that there are no such circumstances when back-tracking happens while a loop has not been detected. However, there is a chance that a loop could not be detected for a reason (e.g., a bug in the implementation, or an immature UI similarity detection, or a UI complexity). If this happens, the discovered app UI will be treated as a new one, and thus it will be added into the f -UFG model. In our technique, the condition for loop detection is based on the mechanism of similarity detection between two UIs. That is, during exploration of the app GUI, the foreground app UI will always be compared with the discovered UIs which already exist in the f -UFG model. If the foreground app UI matches one of the app UIs from f -UFG, the flag will be raised that the loop has been detected, and then back-tracking happens.

4.4 Design and Implementation of DroidDEV: Practical Experience

In this section, we provide the details of DroidDEV practical experience which are learned its during design and development phases. For that purpose, we specify several important questions to answer. The questions concern (1) an automated GUI traversing via the f -UFG model using Best-first search (A^*),⁷ and (2) an automated generation of the relevant user inputs via **UI-context-aware input generation** using a UI-context-aware, user predefined, or random values. We list the questions below as follows:

1. What is the f -UFG model?
2. How do we build the f -UFG model?
3. How do we represent the f -UFG model?
4. How do we dump and exercise the UI widgets?
5. How do we implement the GUI traversal technique using A^* ?
6. How do we generate the UI-context-aware textual user inputs?

⁷The A^* search is the best-known form of the *Best*-first search (BFS) algorithm [102].

4.4.1 What is the *f*-UFG model?

In practice, we design DroidDEV to build a such GUI model which is always finite state and without loops. To ensure that the model is a finite state, DroidDEV implements a mechanism of identifying an equivalence of two UIs. To ensure that the model is built without loops, DroidDEV does not memorise a transition from an exercised UI widget, if the performed action on the UI widget leads to an app UI which currently exists (i.e., it is already added) in the *f*-UFG model. So, DroidDEV avoids loops in a GUI exploration by building the *f*-UFG model which consists of a set of unique app UIs which are organised in a GUI-tree hierarchy according to the transitions discovering new app UIs (i.e., app UIs which do not exist in *f*-UFG at this point).

4.4.2 How do we build the *f*-UFG model?

To build the *f*-UFG model, upon exercising every UI widget, DroidDEV adds a subsequent rendered app UI in the *f*-UFG model and memorises a transition only if it leads to a newly discovered app UI, i.e., an app UI which does not exist in the *f*-UFG model. It is important to note that at a time, all the app UIs in the *f*-UFG model are unique, i.e., only one instance of the discovered app UI can exist in *f*-UFG. Once the app UI is fully exercised (i.e., all its relevant UI widgets are exercised), and there are no more subsequent app UIs with unexercised UI widgets, DroidDEV removes the fully exercised app UI from the *f*-UFG model. Note that DroidDEV allows the exercising procedure to re-exercise the previously discovered app UIs multiple times depending on an app GUI structure so the previously discovered app UIs can be added in, and removed from *f*-UFG multiple times.

4.4.3 How do we represent the *f*-UFG model?

To represent the *f*-UFG model in memory, we create the Java objects of classes `UIScreen` and `UIWidget`. In memory, an object of `UIScreen` represents a hierarchy view of an app UI, and an object of `UIWidget` represents a UI widget on an app UI. DroidDEV views the `UIScreen` object as a container, which consists of a set of `UIWidget` objects with all their attributes. In turn, every `UIWidget` object contains a list of the `UIScreen` objects corresponding to their subsequent app UIs which have been discovered upon exercising the app UI widgets which correspond to their `UIWidget` objects. So, the *f*-UFG model has the following hierarchy structure: one `UIScreen` object contains many `UIWidget` objects, and every `UIWidget` object contains a list of subsequent `UIScreen` objects which are discovered upon exercising corresponding app UI widgets.

4.4.4 How do we dump and exercise the UI widgets?

To perform all necessary manipulations with app UIs, we use the UI Automator testing framework. In Java, we extend relevant classes from the UI Automator and use them to manipulate app UIs and all their UI widgets via the corresponding objects of the `UIScreen` and `UIWidget` classes. In turn, we store an app UI with all its UI widgets in the Java memory using the corresponding objects of the `UIScreen` and `UIWidget` classes. Whenever DroidDEV needs to find a relevant UI widget on the current app UI to exercise, it scans through a set of the `UIWidgets` in the `UIScreen` which represents the current app UI, and checks the values of the attributes (e.g., “*editable*” for “*EditText*” widget, or “*clickable*” for “*Button*”) of its UI widgets via accessing their corresponding `UIWidget` objects. For example, to ensure that all inputs are provided, DroidDEV first scans all editable widgets, e.g., “*EditText*”, to inject relevant inputs, and next, it finds a relevant non-editable widget to exercise, e.g., “*Button*”. Important to note that DroidDEV allows re-exercising of the same UI widgets multiple times on the previously discovered app UIs. For example, an app UI has widgets, traversing of which generates dependent app UIs, where every subsequent app UI depends on its previous one. Also, assume that at a certain point in the app GUI, the dependent app UIs form a loop among themselves. So, due to such the loop existence in the app GUI, to exercise all the UI widgets on the dependent app UIs, DroidDEV has to re-visit the previously discovered app UIs several times. In turn, to re-visit the dependent app UIs, DroidDEV has to traverse the same UI widgets multiple times. As such, DroidDEV will generate the relevant user inputs, and UI events on the respective app UIs as many times as needed, until all the UI widgets on all the dependent app UIs are fully exercised.

4.4.5 How do we implement the GUI traversal technique using A*?

In practice, DroidDEV implements an automated GUI traversing of the mobile apps using the existing A* technique. In fact, due to the complex nature of the mobile apps GUI structure, it is not practical that the modern mobile apps GUIs consist of only one single UI. Our observations show that a loop usually occurs during an exercising procedure for all but trivial apps GUIs. So, if the loop has occurred, we need DroidDEV to do traversing in *f-UFG* in order to return to the lastly discovered app UI which DroidDEV has left during exercising. In particular, to find a path from an app UI to the lastly discovered one in the *f-UFG* model, DroidDEV uses an execution trace from the exercised UI widgets. During the exercising procedure, DroidDEV records the execution trace following an innate GUI execution path (transitions). Along the GUI execution path, DroidDEV marks all the exercised UI widgets with flag ‘1’ and memorises transitions from the exercised UI widgets to their subsequent app UIs. If during exercising, the current app UI on the Android device screen is changed to any previous one which is already added

in *f-UFG*, DroidDEV performs the UI search starting from that previous app UI. In particular, for every app UI, DroidDEV recursively searches for the UI widgets satisfying two conditions, where (1) the UI widgets are marked with flag ‘1’ (i.e., exercised), and (2) the UI widgets have memorised transitions leading to their subsequent app UIs. If both conditions are satisfied, DroidDEV performs a relevant action, e.g., “Click”, upon a corresponding UI widget to switch to its subsequent app UI according to the transition. DroidDEV repeats the UI search until the lastly discovered app UI is reached, and becomes the current app UI displaying on the Android device screen. Next, DroidDEV continues GUI exploration from the current app UI by exercising its retaining widgets.

4.4.6 How do we generate the UI-context-aware textual user inputs?

A **UI-context-aware input generation** mechanism encompasses two types of the generated user inputs: *UI-context-aware*, and *Random/Default*. For each input generation method, we set a priority. So, *UI-context-aware* as the first method to perform, and *Random/Default* is the second method to perform. Note that once any of the methods return a generated input, DroidDEV stops the inputs generation process, and uses the returned input in the exercising process. Defining such sequence, DroidDEV finds the most relevant textual inputs to the current app UI. In the following sections, we describe each of the methods and explain how they are implemented in practice.

UI-context-aware textual user input generation

DroidDEV uses the *UI-context-aware* input generation method to obtain input values which lie in a set of possible values with a specific input pattern required. For example, email address, or IP/URL of the server. All of such inputs may have different actual values. However, valid ones can only be the inputs with a correct pattern. For example, for an email address, it can be ‘*email@host.com*’, and for a server address, it can be ‘*88.14.36.11*’ or ‘*xxx00053.outlook.com*’. These values have a predefined pattern which can only be recognised if a valid pattern of the email address or valid pattern of the server address is used. So, if the email address is required by an app, a user should not provide any random characters without a specific pattern of the email, e.g., ‘*aaa.email.com*’ will not be recognized as a valid email so that the app is unlikely to proceed with such the input even the app does not do any input validation. Therefore, such invalid user inputs may block a testing procedure, and thus human interaction will be required. So, the *UI-context-aware* inputs are designed to resolve such cases and automate the testing process. In practice, we found that most commonly used basic terms are ‘*password*’, ‘*login*’, ‘*email*’, ‘*e-mail*’, ‘*server*’, ‘*port*’, ‘*url*’, ‘*phone*’, ‘*account*’, ‘*id*’, ‘*name*’, ‘*number*’, ‘*tel*’, and ‘*mobile*’. For these text terms, we create a dictionary with pairs ‘*key-value*’, where the terms serve as a ‘*key*’, and the default values with correct pattern serve as

a ‘value’. Note that our dictionary does not allow ‘key-value’ pairs to be duplicated, i.e., for the same ‘key’ only one ‘value’ is allowed at a time (i.e., per testing app). For example, if the pair ‘*email = mobolic@gmail.com*’, where ‘*email*’ is the ‘key’, and ‘*mobolic@gmail.com*’ is the ‘value’, has already been added to the dictionary, no other email address can be added, i.e., it is only one record with ‘*email*’ is allowed. As such, the dictionary will not contain multiple email addresses for one testing app. The same rule applies to other *keys* existing in the dictionary. In practice, our dictionary consists of the input types and their corresponding default valid input values. For example, for an editable UI widget which requires *email* input type, the dictionary defines a default valid input value “*email@example.com*”. However, optionally, to simulate a real-case scenario, a user may create an actual email, e.g., “*mobolic@gmail.com*”, and provide this email address to DroidDEV by setting the field ‘*email*’ in the project “*config.properties*” file so that DroidDEV will automatically replace the default email address by the user provided one.

To summarise, we stress that DroidDEV uses the *UI-context-aware* input generation method to automatically find which input type is required for a particular editable widget, and to generate either predefined or user-provided inputs corresponding to its found input type. As such, a user does not need to be involved in a testing process. Instead, the user may provide actual values for the most common input types such as ‘*email*’, ‘*password*’, ‘*login*’, ‘*server*’, ‘*port*’, ‘*url*’, and ‘*phone*’ in the project “*config.properties*” file only once, before an exercising process starts. So, during exercising, DroidDEV finds the required input type and injects its corresponding value into the relevant editable UI widget.

Random and default textual user input generation

We allow Random/Default inputs to ensure that DroidDEV does not require any human interaction in a testing process. In case, when inputs are not found by the *UI-context-aware* method, DroidDEV will take a default or random value (numeric or string). DroidDEV first checks if an editable UI widget contains a default value (e.g., actual value or hint in the editable widget) which is provided by the testing app itself. If so, DroidDEV memorises this value for a future input generation. If the editable UI widget is empty (no default value or hint), for random input, DroidDEV sets a predefined textual value ‘*defaultValue*’ or generates a random integer value in a range [1, 100]. Depending on the app, if a numeric value is rejected (e.g., only string values are allowed in the editable UI widget), DroidDEV injects a predefined textual value.

4.5 DroidDEV: Demo

Next, we illustrate an overall workflow of DroidDEV through an example of an app. The UFG model of the demo app is given in Figure 4.1, Section 4.3.3. The app consists of five app UIs (screens). DroidDEV exercises the app through the following steps:

1. Run the app and obtain the first UI, i.e., u_1 . Add u_1 and its widgets to the f -UFG model. SensINJECT injects input into e_{11} . Afterwards, DynaEXE clicks the first unexercised non-editable widget w_{12} so u_2 is rendered. Thus, DynaEXE adds the link (w_{12}, u_2) and u_2 with its widgets to the f -UFG model and it returns TRUE to indicate that there is no disruption in the execution flow. The current UI is now u_2 and the execution flow continues.
2. Repeat the previous step for u_2 and u_3 . After exercising w_{32} , the current UI is u_4 and the execution flow continues.
3. On u_4 , DynaEXE clicks w_{41} , thus rendering u_1 . As u_1 is already in the f -UFG model, the edge (w_{41}, u_1) will cause a loop in the f -UFG model. Therefore, DynaEXE will not add the edge (w_{41}, u_1) into the f -UFG model. DynaEXE returns FALSE to indicate that there is a disruption in the execution flow (line 14, Algorithm 1 in Section 4.3.4, *continuity* equals to FALSE). DroidDEV updates the current UI to u_1 and BestPath finds the best path to the lastly discovered UI, i.e., u_4 . The best path includes $(u_1, w_{12}, u_2, w_{21}, u_3, w_{32}, u_4)$. DroidDEV retraces this path. After retracing the best path, the current UI is u_4 .
4. On u_4 , DroidDEV retrieves the next unexercised, non-editable widget from u_4 , i.e., w_{42} . DynaEXE clicks w_{42} , thus rendering u_4 . Since u_4 is already in the f -UFG model, the edge (w_{42}, u_4) will not be added into the f -UFG model. DynaEXE returns FALSE to indicate a disruption in the execution flow. After exercising w_{42} , u_4 has no unexercised widgets and thus u_4 is fully exercised. As such, DroidDEV adds u_4 with its widgets into a set of the covered UIs and removes it from the f -UFG model. As a result, in the f -UFG model, DroidDEV changes the lastly discovered UI to u_3 . DroidDEV also realizes that the current UI is u_4 which has been removed from the f -UFG model and added into the set of the covered UIs. Therefore, DroidDEV generates the system “Back” event (line 22, Algorithm 1 in Section 4.3.4). The “Back” event brings the app to any previously discovered UI. Without loss of generality, we assume that the event brings the app back to u_1 . DroidDEV updates the current UI to u_1 and BestPath finds the best path from u_1 to the lastly discovered UI in the f -UFG model, i.e., u_3 . The best path includes $(u_1, w_{12}, u_2, w_{21}, u_3)$. DroidDEV retraces the best path and updates the current UI to u_3 .

5. On u_3 , DroidDEV retrieves the next unexercised, non-editable widget from u_3 , i.e., w_{33} . DynaEXE clicks w_{33} , thus rendering u_5 which is not in the f -UFG model. Thus, DynaEXE adds the link (w_{33}, u_5) and u_5 with its widgets to the f -UFG model, and returns TRUE to indicate that there is no disruption in the execution flow. DroidDEV updates the current UI to u_5 and the execution flow continues.
6. On u_5 , SensINJECT injects inputs into editable widgets e_{51} and e_{52} . Afterwards, DynaEXE clicks the first unexercised non-editable widget w_{53} , thus rendering u_3 . As u_3 is already in the f -UFG model, the link (w_{53}, u_3) is not added to the f -UFG model. DynaEXE returns FALSE to indicate a disruption in the execution flow. After clicking w_{53} , all the widgets on u_5 are exercised. As such, u_5 with its widgets is added into the set of the covered UIs and removed from the f -UFG model. As a result, the lastly discovered UI in the f -UFG model is changed to u_3 . DroidDEV updates the current UI to u_3 .
7. Since all the accessible widgets on u_3 have been exercised, DroidDEV adds u_3 with its widgets into the set of the covered UIs and removes it from the f -UFG model. As a result, in the f -UFG model, DroidDEV changes the lastly discovered UI to u_2 . Note that the current UI is u_3 which has been removed from the f -UFG model and added into the set of the covered UIs. As such, DroidDEV generates the system “Back” event to u_1 . Afterwards, BestPath searches for the best path from u_1 to the lastly discovered UI in the f -UFG model, i.e., u_2 . The best path includes (u_1, w_{12}, u_2) . DroidDEV retraces the best path and updates the current UI to u_2 .
8. The previous step is repeated for u_2 and u_1 and these two UIs with their widgets will be removed from the f -UFG model and added into the set of the covered UIs.
9. At last, the f -UFG model is empty and DroidDEV terminates its exercising procedure. All the accessible widgets on all the discovered UIs from the f -UFG model have been exercised.

4.6 Empirical Evaluation

In this section, we explain how we do a setup for our experiment to compare the performance of DroidDEV with MobiGUITAR [2], Monkey [95], and ManualTest (manual UI testing by a tester). MobiGUITAR is a tool for automated model-based testing of the Android mobile apps. It implements both systematic exploration strategies (i.e., Depth-first and Breadth-first) and random one. Monkey is a fully automated tool which generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as certain system-level events. Monkey treats an app under test as a black-box. It exercises the app by simply performing random actions at random positions on the app

UI. Even though Monkey is simple, it remains state-of-the-art and state-of-the-practice an automated technique. Based on the paper’s results [15], Monkey outperforms many existing prevalent automated testing tools on average. Due to this reason, we use Monkey as a “gold standard” for the performance comparison with our technique. As such, we believe that comparison with Monkey creates a reliable picture of the validity of our technique results.

4.6.1 Experimental setup

From F-Droid,⁸ we downloaded 20 real-world open-source Android apps from different categories. For our experiment, we downloaded more apps from the “*Games*” and “*Multi-media*” categories because these apps are generally highly interactive and challenging to test automatically. It is also important to note that most of the code of the downloaded apps is UI-related. Therefore, we ensure that the obtained code coverage results correctly show an ability of DroidDEV to exploring the apps GUIs via an execution of the UI-related code by interacting with their GUI elements (since our testing technique is GUI-based) rather than any other app code, e.g., system-related code by triggering the system events which are not supported by DroidDEV. In addition to the apps selection procedure, since our technique is primarily designed to demonstrate how exhaustively it can exercise the discovered app UIs via the code coverage measurement, we excluded those apps which happen to have an “unnecessary code” which surely cannot be executed by our technique firing the UI events. For example, (1) a system-related code (via broadcast receivers) which are triggered by receiving system events, however, such events are not supported by our technique; or (2) a platform version-dependent code, portions of which can only be executed on a specific version of the Android platform, or (3) a “dead code” which has been left in the app (could be a bug or forgotten to be removed), and thus will never be executed. Considering such criteria for the apps exclusion from our experiment, we believe that we do not bias our results, and we do not favour our technique in any way. For further assurance of the validity of our results, we use the same set of the selected apps to compare the results of our technique with others such as ManualTest, Monkey, and MobiGUITAR. Using the same set of apps, we demonstrate a capability of these tools via code coverage results how exhaustively they can explore an app GUI comparing to our technique.

We run DroidDEV, MobiGUITAR and Monkey for all 20 Android apps on eight Android emulators, and average the obtained code coverage results for the performance evaluation. Since DroidDEV is self-terminative, we do not limit its exercising time. However, for MobiGUITAR and Monkey, we need to set a time limit to terminate their exercising procedures since both tools are not self-terminative. Based on the

⁸<https://f-droid.org>

recommended configuration, we set 60 minutes exercising time limit for each app for MobiGUITAR.⁹ To fairly compare MobiGUITAR with Monkey, we set 60 minutes for each app for Monkey. Note that settings for Monkey are different from MobiGUITAR. For Monkey, the number of events to be injected must be specified. Once the specified number of events has been injected, Monkey terminates its exercising process. In this experiment, we set the number of events to be 5,000,000 to enable Monkey continuously run for at least 60 minutes for each app. Once 60 minutes have elapsed, we kill the Monkey process.

To perform manual testing (i.e., ManualTest), we selected eight advanced Android users with knowledge in software testing on the Android platform. They know the intricacies of the Android app UI testing. We assigned all 20 apps to all eight advanced users and averaged their code coverage results. To make a fair comparison with DroidDEV, we required each user to exercise each app until he/she still can discover any new app UI (i.e., it is according to his/her understanding about the new app UI) in order to ensure that the maximal possible code coverage was achieved.

4.6.2 Code coverage analysis with EMMA

What instruments do we use for code coverage generation and collection?

For code coverage generation and collection, we use the EMMA tool.¹⁰ EMMA is an open-source tool for measuring and reporting the Java code coverage, and it is included in the Android SDK by default. EMMA can instrument the Java bytecode (*.class) for coverage either offline (before *.class files loaded), or on-the-fly (when the Java bytecode is being executed). EMMA supports coverage types such as “class”, “method”, “basic block”, and “line”. Note that “line” coverage is not provided without source code being available. In addition, EMMA supports three types of coverage reports in *.txt, *.html, and *.xml file formats.

There are two ways to use EMMA with the Android apps: (1) to include EMMA (pre-dexed form of *emma.jar*) into the Android system image (*system.img*), or (2) to include EMMA into the *classes.dex* file of an app (*.apk). We chose (2), which is easier, and more universal way, and allows us to run apps on multiple Android platforms (if needed). Note that the pre-dexed form of *emma.jar* is not enough to make EMMA work on the Android platform. The EMMA’s folder structure with their content must also be included in the root of the testing *.apk file (the same folder level as *classes.dex*).

⁹bear.cc.gatech.edu/~shauvik/androtest

¹⁰<http://emma.sourceforge.net/faq.html>

Which coverage metric (measure) for executed Java code do we use?

The code coverage is the most useful indicator showing the effectiveness of a testing approach, simply because the lower coverage results, the more app code remains untested. EMMA suggests that “*basic block*” is a fundamental coverage metric, and other coverage metrics such as “*branch*”, “*class*”, “*method*”, and “*line*” can be derived from “*basic block*”. As such, in our experiment, we use the “*basic block*” coverage metric for evaluation of our code coverage results. The “*basic block*” is a well-defined metric, and it is represented as a sequence of bytecode instructions without any jumps or jump targets inside, i.e., as an atomic unit.

EMMA marks a “*basic block*” as covered once the control has reached its last instruction. Therefore, the “*basic block*” is guaranteed to have been executed without failure at least once, and reflects more precisely how adequately app functionality was exercised, rather than “*lines*”, “*classes*”, or “*methods*”. Note that the “*line*” coverage metric is a poorly defined concept and should not be used as a code coverage metric except when linking line coverage to the original source code. In contrast, the “*class*”, and “*method*” metrics are well-defined, but indeed cannot be used to show how adequately app functionality was tested. In particular, EMMA marks the Java “*class*”, or “*method*” covered once the Java class is loaded by the Dalvik VM or Java VM (depending on the running virtual machine), or method is entered (very first “*basic block*” of the method is covered). Instead, the “*class*” and “*method*” coverage metrics should be used to detect, e.g., “*dead code*”, in the app, rather than showing the performance regarding an achieved code coverage.

How do we collect code coverage data?

To dump (collect) code coverage, we send an activity broadcast intent¹¹ to an app under test. To have unique intent names for each app, we use their package names appended with “*.EMMA_DUMP_COVERAGE*”. Once such an intent has been sent, it calls the *onReceive(...)* method from our Java class, namely “NxAe85AOZo9UAYV” for the reasons of uniqueness, which extends *BroadcastReceiver* Java class. The Java class “NxAe85AOZo9UAYV” is included in the *classes.dex* of the testing **.apk* where we dump the Java code coverage by calling EMMA’s method *dumpCoverageData(...)*. Note that since we include EMMA in **.apk*, we are able to call EMMA methods directly without applying the Java reflection mechanism.

To let an app under test react to sent broadcast intent, there are two ways: (1) to statically register a broadcast receiver in the *AndroidManifest.xml* file, or (2) to dynamically register a broadcast receiver at runtime. We chose (1) as it is a more reliable, and less error-prone method. Note that in method (1) all receivers are registered

¹¹<https://developer.android.com/guide/components/broadcasts.html>

globally, in contrast to (2) where receivers can be registered globally or locally. If method (2) is selected, in order to register a receiver globally, the Android app context should be used, i.e., `getApplicationContext().registerReceiver(...)`. We use (1) to register our broadcast receiver globally since the Java code coverage results are needed for the entire app (from all possible app states and UIs). If a broadcast receiver is registered locally (e.g., for a specific app activity/UI), coverage results can be collected only for the activity with its corresponding (*.class) code while it is running in the foreground.

To make sure that all the coverage data is collected, after every performed action on the app UI, we dump an intermediate code coverage (*coverage.ec*) into the app under test folder on the emulator, i.e., `"/data/data/app_package_name/files"`. We use the default app's folder on the emulator to avoid making other changes to the *AndroidManifest.xml* file (e.g., adding extra permissions to allow the app to write the code coverage files on external storage). Note that the coverage data will be lost if the app has crashed. Therefore an intermediate code coverage is necessary to obtain the coverage results for any previously executed Java code.

How do we identify executed Java code?

We identify the executed Java code via an app code instrumentation using the EMMA tool. To instrument the app *.class files, we use the offline EMMA method. First, we generate Android *.apk files from the source code using the Eclipse IDE for Android. For instrumentation, we consider only *.class files which correspond to the source code *.java files. However, we exclude auto-generated *.class files such as *"BuildConfig.class"* and all *"R*.class"* files from the EMMA instrumentation since they would contribute to the code coverage with "0" coverage values for all the coverage metrics since they are not executable. Once all the app *.class files are instrumented, EMMA generates one *"coverage.em"* file with all the metadata required for the code coverage report generation.

4.6.3 Experimental results

In Table 4.2, we show the *"basic block"* app code coverage obtained by DroidDEV, ManualTest, MobiGUITAR and Monkey (columns 3–6) and the exercising time taken by DroidDEV in the last column. From the last column in Table 4.2, we can see that the average exercising time taken by DroidDEV for each app is equal to 15.76 minutes which is much shorter than the set time limit (60 minutes for each app) for MobiGUITAR and Monkey. The exercising time for DroidDEV varies depending on the GUI model of the app under test. The minimum time taken by DroidDEV is 0.35 minutes for the *Core* app, and maximum one is 117.82 minutes for the *MemoPad* app. The *Core* app is a game consisting of one single main UI requiring complex sequences of the UI events leading to

win or lose the game. In turn, the *MemoPad* app takes much longer since it has a more complex app GUI model consisting of multiple distinct UIs with a variety of widgets.

The coverage data given in Table 4.2 (columns 3–6) is computed as follows:

$$Coverage = \frac{Average_number_of_covered_basic_blocks}{Total_number_of_basic_blocks}$$

As for ManualTest, the average number of the covered basic blocks is computed by averaging the code coverage results given by eight manual testers. As for DroidDEV, MobiGUITAR and Monkey, the average number of the covered basic blocks is computed by averaging the code coverage results given by eight Android emulators.

DroidDEV vs. ManualTest

Table 4.2 shows that for most the apps, DroidDEV performs similar to ManualTest. However, there are some noticeable differences in the code coverage of several apps, which are due to their specific nature. For example, DroidDEV shows a better performance for the *Ringtone Generator* and *Mytronome* apps. We manually studied and found that these apps require more comprehensive exercising with long sequences of the UI events, which DroidDEV, being an automated tool, can generate more efficiently.

For the apps such as *Core*, *DivideAndConquer*, and *MunchLife*, ManualTest shows a better performance. It is simply explained by the fact that humans are better at providing short and more intelligent sequences of UI events than automated approaches. Furthermore, these three apps are all games and humans can better deduce the complex sequences of UI events required to win or lose the game. For the remaining apps, DroidDEV and ManualTest perform similarly. From Table 4.2, we can see that DroidDEV covered 91.5%, and ManualTest covered 91.0% of each app code on average. Thus, DroidDEV is capable of achieving high code coverage.

DroidDEV vs. MobiGUITAR

Table 4.2 also provides the code coverage results for MobiGUITAR. It shows that, on average, MobiGUITAR achieves low (44.9%) code coverage. After our thorough manual investigation, we found that such results are due to the lack of support for the complex UI events in MobiGUITAR, e.g., motion and trackball UI-events. Instead, MobiGUITAR generates standard UI events such as “Click”, and “LongClick”, “Input Injection”, hardware keys, and system events such as “Home”, “Back”, and “Menu”. However, many of the selected apps such as *AndroFish*, *DivideAndConquer*, *Battery Dog*, *Acrylic Paint*, *Coloring for Kids*, and *Dotty* mainly rely on the complex user events such as motion (e.g., “Swipe” and “Scroll”), and trackball events (e.g., “Roll” and “Press”). As such, MobiGUITAR could not exercise the app functionality adequately. Even with a

Table 4.2: Code coverage for DroidDEV, ManualTest, MobiGUITAR, Monkey, and time taken by DroidDEV

App category	App name	DroidDEV, %	ManualTest, %	MobiGUITAR, %	Monkey, %	DroidDEV, mins
Development	Dotty	100	100	55.9	100	1.21
	MunchLife	79.6	88.7	33.9	55.7	1.61
	AndroFish	95.2	91.5	4.0	42.8	10.87
Games	Core	85.1	92.5	53.2	88.2	0.35
	DivideAndConquer	86.2	92.5	41.5	78.4	4.69
	GM Dice	81.8	77.9	31.3	55.6	44.52
Graphics	Coloring for Kids	96.6	96.7	48.7	94.7	37.04
	Acrylic Paint	94.6	95.8	18.3	61.2	5.42
	Clock Live Wallpaper	94.4	95.1	1.7	4.8	0.62
Multimedia	Ringtone generator	90.6	77.4	6.7	51.7	24.29
	Signal Generator	98.9	97.3	78.6	96.4	8.95
	Mytronome	98.2	86.6	90.1	91.3	2.94
	Instant Sound Effects	99.3	97.4	78.3	95.8	7.73
Security	Lock Pattern Generator	87.6	90.9	40.8	74.3	5.03
System	Battery Circle	86.9	89.0	51.3	58.2	1.69
	Battery Dog	88.2	86.8	12.4	35.2	12.24
Time	World Clock	98.1	97.1	88.2	95.2	16.12
	Colour Clock	97.0	95.2	91.8	98.2	3.45
Writing	MemoPad	85.4	85.6	47.6	86.5	117.82
	WikiAndPad	86.7	85.9	23.7	72.9	8.67
Average		91.5	91.0	44.9	71.9	15.76

given much longer exercising time of 60 minutes for each app as compared to the time taken by DroidDEV (15.76 minutes on average), on average, MobiGUITAR failed to achieve high code coverage.

DroidDEV vs. Monkey

Table 4.2 also provides the code coverage results for Monkey. On average, DroidDEV outperforms Monkey. It can be seen that the results obtained by Monkey vary from very low (4.8%) to very high (100%). Such the code coverage results can be simply explained by the Monkey’s nature, i.e., Monkey generates random events at random positions on an app UI. Therefore, its code coverage results may significantly vary from app to app, and be affected by a density and physical size of the UI widgets. Thus, on average, Monkey shows lower coverage results compared to DroidDEV. In particular, DroidDEV covers 91.5%, while Monkey covers 71.9% of each app code on average. Given a much longer exercising time of 60 minutes for each app as compared to the time taken by DroidDEV (15.76 minutes on average), on average, Monkey failed to achieve high code coverage.

4.7 Discussion

In this section, we discuss principal differences between DroidDEV and traditional model-based testing. We describe novel aspects brought by DroidDEV into automated testing of the mobile apps.

DroidDEV uses the model-based testing technique to systematically explore an app GUI by generating relevant events “on-the-fly” which are immediately executed. Using the model-based testing technique, it benefits DroidDEV to potentially execute tests for a long time so that long, intricate, or stressful tests can be performed, and significantly reduce the state-space (i.e., size of the GUI model) to be stored since only a small portion of the GUI model need to be stored at any point in time. The GUI model is automatically extracted from an app and incrementally updated during the testing procedure. In particular, DroidDEV dynamically builds and updates the *f-UFG* model to ensure that the exercising procedure never falls into an infinite loop (since *f-UFG* has no loops), and is always able to finish the testing procedure (since *f-UFG* has a finite number of the app UIs). DroidDEV builds the *f-UFG* model from the discovered app UIs, which is further used to explore another yet undiscovered app UIs.

To efficiently explore an app GUI, DroidDEV implements A*, an informed search algorithm, which is the best-known form of *Best-first* search (BFS) [102, 183]. The A* search algorithm combines BFS for efficiency with the uniform cost search for optimality and completeness. The key idea behind the A* search algorithm is to efficiently generate the shortest sequences of the UI events leading to yet unexplored app UIs. It is

worth noticing that DroidDEV performs the A^* search in the $f\text{-}UFG$ model. Therefore, DroidDEV always generates the shortest sequence of the UI events leading to the lastly discovered, fully unexplored app UI. In addition, in $f\text{-}UFG$, there is always at least one sequence of the UI events leading to the lastly discovered app UI. Therefore, DroidDEV guarantees that such the sequence of the UI events is always generated.

In sharp contrast, existing model-based testing tools [15] commonly implement a uniformed (standard) *Depth*- or *Breadth*-first search algorithms [183] as a standalone (i.e., either one) or their combination is used for automated GUI exploration. However, in practice, their implementation of *Depth*- or *Breadth*-first search algorithms does not demonstrate the sufficient efficacy to adequately explore an app GUI [15, 16]. As a fact, due to the nature of the mobile apps GUIs, their models commonly have an infinite number of app UIs, and/or loops in their GUI models. Therefore, existing tools [15, 16] with their *Depth*- or *Breadth*-first search algorithms implementation may fall into such loops, or find an infinite number of app UIs preventing their exercising procedure from an automatic termination. So, in order to adapt the tools to practice and enable them to automatically finish the exercising procedure, a user is required to manually set a termination condition, e.g., limit the exercising time, or limit the overall number of events to be injected, or limit the depth of GUI exploration. As a result, such limitations may deny the automated tools from performing adequate GUI exploration. However, in the case of a finite state app GUI model without loops, using their *Depth*- or *Breadth*-first search implementation, the tools may effectively explore the app GUIs.

In summary, DroidDEV is capable of exploring app UIs at a deep depth of an app GUI model while avoiding falling into an infinite loop or discovering an infinite number of app UIs. Therefore, DroidDEV addresses the problem stated in Section 4.3.1 by building the $f\text{-}UFG$ model, and the performing A^* search in the $f\text{-}UFG$ model. In contrast, existing tools have limited exploration capability since they perform either *Depth*- or *Breadth*-first search in the innate app UI model which is used “as is” for systematic GUI exploration.

4.8 Conclusion

We have implemented the DroidDEV tool, an automated app UI exerciser, to achieve high code coverage in automated functional UI testing of the Android apps. It dynamically refines the finite UI-flow graph ($f\text{-}UFG$) of an app under test, injects the UI-context-aware inputs into the editable UI widgets, and generates the UI events to automatically exercise app UIs. DroidDEV implements the Best-first search algorithm to effectively and efficiently discover all the app UIs and exercise all their widgets. Our experimental results show that DroidDEV performs similarly to manual UI testing regarding the code coverage. More importantly, DroidDEV outperforms prevalent existing automated approaches such as MobiGUITAR and Monkey regarding the code coverage and exercising time.

Chapter 5

Automated Textual Input Generation for Functional Testing of Mobile Apps GUI

5.1 Summary

Increasingly prevalent use of mobile devices has raised the popularity of mobile applications. Therefore, automated testing of mobile applications has become a crucial task. However, it is still a challenge to automatically generate tests with high coverage for mobile applications due to their specific non-trivial structure, and highly interactive nature of the user interfaces. In this chapter, we propose an enhanced version of DroidDEV,¹ another automated GUI testing technique for mobile applications, namely *Mobolic*. It improves DroidDEV by using more sophisticated input generation mechanism which additionally involves dynamic symbolic execution for a concrete user input generation. From the other side, *Mobolic* uses the same GUI-model construction based on the *f-UFG* model, and the *UI-context-aware* input generation mechanism as in DroidDEV.

In *Mobolic*, tests with high coverage are automatically generated and executed by combining the online testing technique,² and customized input generation.³ Employing the online testing technique, *Mobolic* systematically explores an app GUI without falling in a loop. It generates relevant events “on-the-fly” which are followed by an immediate execution. In addition, involving the customized input generation, *Mobolic* automatically generates relevant user inputs such as user predefined, concrete or random ones. We implemented *Mobolic* and evaluated its performance on 10 real-world open-source Android applications. Our experimental results show the effectiveness and efficiency of *Mobolic* regarding the achieved code coverage and overall exercising time.

¹The DroidDEV technique is described in Chapter 4.

²The online testing technique is implemented in DroidDEV, and exploited in *Mobolic*.

³The customized input generation is newly introduced in *Mobolic*.

5.2 Introduction

In the last few years, there has been a distinct shift to mobile devices in numerous application areas such as e-mail, social networking, entertainment and e-commerce. This trend has prompted an explosive growth in the number and variety of mobile applications (often called “apps”) are being developed. As such, there is an increasing demand for automated testing techniques for mobile apps.

Mobolic aims to automatically generate tests with high coverage via automated exercising of the mobile apps GUIs with customized user inputs. It does this by integrating the online testing technique [16,46,100] and customized input generation into a single solution. It uses (1) the model checking technique and finite state machine abstraction for a systematic exploration of the app GUI via an automated guiding of the exercising process, and (2) the customized input generation which involves the UI heuristics such as textual values of the editable UI widgets’ attributes to automatically generate relevant user predefined or random inputs, and involves symbolic execution to generate concrete user-inputs.

Mobolic automatically explores an app GUI by systematically interacting with the app UI widgets using our DroidDEV technique which is described in Chapter 4. However, due to the highly interactive nature of an app GUI, it commonly requires specific user inputs which are, practically, impossible to generate randomly. So, to generate such specific user inputs, *Mobolic* implements our novel customized input generation mechanism. It allows *Mobolic* to generate realistic user inputs which are predefined by a user for certain input types. Also, customized input generation mechanism allows *Mobolic* to generate concrete user inputs. For that reason, *Mobolic* uses the dynamic symbolic execution technique. However, there are situations when user-specific or concrete user-inputs cannot be generated, and thus to fully automate the exercising process, *Mobolic* supports a random input generation. As a result, *Mobolic* explores an app GUI without the human interaction.

In order to perform automated testing within a reasonable amount of time for different apps, we designed *Mobolic* as a separate technique from DroidDEV. These two techniques are meant to be used for different kinds of apps. In particular, *Mobolic* involves dynamic symbolic execution which is slow in nature. As such, it slows down *Mobolic* exploration GUI speed resulting in longer total execution time. Therefore, from a practical point of view, it will be more reasonable to use *Mobolic* for the apps which heavily rely on specific user inputs, where a testing procedure can benefit from symbolic execution. Even though *Mobolic* can also be used for apps which accept any user input, however, a testing process may not benefit much from symbolic execution while inevitably prolonging the overall execution time. As such, using *Mobolic* for the apps which accept any user inputs is still possible. However, it is not recommended from a practical point of view since

another lightweight technique, DroidDEV, can achieve similarly high code coverage while taking reasonably less time for such kinds of apps. So, having DroidDEV and *Mobolic* as two separate techniques, makes it more flexible for a user to choose the most suitable technique depending on the target apps to be tested.

In this chapter, we made the following contributions:

- We propose a novel customized input generation mechanism in our enhanced automated GUI testing technique, namely *Mobolic*. It uses advances of our DroidDEV technique, and novel customized input generation for generating tests with high coverage.
- We implement *Mobolic* for the Android apps, taking into account their highly interactive nature of GUIs.
- We evaluate the performance of *Mobolic* on 10 real-world open-source Android apps, and compare it with prevalent approaches such as manual exercising (Human), Sapienz, ACTEve and MobiGUITAR.

In Section 5.3, we state the main problem and describe the proposed approach by giving definitions, overview, and detailed explanations of *Mobolic* components. In Section 5.4, we describe design and implementation of *Mobolic* by sharing our practical experience which is obtained during the *Mobolic* development. In Section 5.5, we give a demo of the *Mobolic* workflow by using a concrete example of the app GUI. In Section 5.6, we evaluate the performance of *Mobolic* by comparing it with prevalent approaches such as Human, Sapienz, ACTEve, and MobiGUITAR. In Section 5.7, we discuss the *Mobolic* approach. We discuss its novel and improved features by describing the principal differences between *Mobolic* and traditional model-based testing and between *Mobolic* and automated input generation via symbolic execution. In Section 5.8, we identify three main threats to the internal and external validity of the obtained results for the code coverage and exercising time. In Section 5.9, we offer an overview of related work by describing the state-of-the-art automated GUI testing approaches such as Monkey, Dynodroid, MobiGUITAR, A³E, *SwiftHand*, Sapienz, and ACTEve. In Section 5.10, we conclude our chapter and identify possible future research directions.

5.3 Proposed Approach: Mobolic

In this section, we describe the related problems, give definitions, overview, and details of *Mobolic*. In Section 5.3.1, we state the main problem which *Mobolic* addresses. In Section 5.3.2, we give several conceptual definitions for the *Mobolic* explanations. In Section 5.3.3, we describe novel aspects of *Mobolic*. In Section 5.3.4, we introduce *Mobolic* by briefly describing its components, and walking through its overall exercising

procedure. In Section 5.3.5, we describe in detail each component of *Mobolic* by explaining their working mechanisms.

5.3.1 Problem statement

The mobile apps GUIs are commonly highly interactive so that the apps require sensible input from users. As such, to automatically exercise the app, it is crucial that non-trivial user inputs must be automatically generated to feed into the app. To the best of our knowledge, this remains a challenging task. For example, many existing tools for the GUI testing of mobile apps either generate inputs randomly and/or provide them manually during testing [2, 10–13, 15, 88, 103, 119, 136]. However, random inputs are trivial, while manual efforts are time-consuming. Therefore, to automatically generate relevant user inputs, existing tools [8, 17, 18] have applied symbolic execution. These tools tend to symbolically execute all program paths existing in the app. However, symbolic execution suffers from limitations such as *path explosion*, *path divergence*, and *constraint complexity* [19]. Therefore, to execute all program paths symbolically is impractical for all but trivial apps. So, the search of relevant inputs needs to be either *depth*-bounded or *time*-capped [20, 21].

5.3.2 Definitions

For the explanations of *Mobolic*, in this section, we give two definitions which are used throughout this chapter.

Definition 1: User-Input-Dependent statement (UID statement) is a statement which is control- and/or data-dependent [184, 185] on one or more input values provided by a user through an app GUI.

An example of code in Figure 5.1 gives an illustration of the UID statements. From Figure 5.1, we can see that the UID statements are in lines 2 and 7, while the non-UID statement is in line 14.

Definition 2: Reduced Control Dependence Graph (r-CDG) is a conventional CDG, where all the decision nodes correspond to UID conditional statements.

In Figure 5.2, we give an example of r-CDG, which is obtained from the app code shown in Figure 5.1. In Figure 5.2, the nodes ①, ②, ④, ⑤, ⑥ correspond to the code lines 2, 3–4, 7, 8–9, and 11–12, respectively, as shown in Figure 5.1. In Figure 5.2, the decision nodes ① and ④ (i.e., “if”-statements) are shaded, and *T* and *F* abbreviations denote “True” and “False” branches, respectively.

```

1: void foo(int x) { /* "x" contains user value */
2:   if (x > 5) {
3:     /* to discover  $u_2, u_3$  */
4:     /* to execute code#1 */
5:   }
6:   else {
7:     if (x == 5) {
8:       /* to execute code#2 */
9:       return;
10:    }
11:    /* to discover  $u_1$  */
12:    /* to execute code#3 */
13:  }
14:  if (getSystemTime() == '00:00:00') {
15:    /* to execute code#4 */
16:  }
17: }

```

Figure 5.1: App code example with UID statements

Figure 5.2: r-CDG example of app code

5.3.3 Novel aspects

In this section, we show the novel aspects of *Mobolic* which are implemented via various improvement(s) and/or new feature(s) which existing tools either do not implement in the same fashion and/or do not have them. In *Mobolic*, we introduce a novel feature such as a ***customated input generation*** mechanism. It enables *Mobolic* to generate concrete user inputs while mitigating the well-known problems of the conventional symbolic execution. The idea is to symbolically execute only UID statements since the nature of the mobile apps' GUIs is highly interactive, and the majority of the textual inputs is, in general, expected from a user. So, it allows *Mobolic* to avoid extra overhead of symbolically executing all existing in an app conditional statements, and thus *Mobolic* finishes the exercising process within a reasonable amount of time.

5.3.4 Overview

In this section, we introduce the components of *Mobolic*, give an overview of the *Mobolic* workflow, and show an overall testing procedure of *Mobolic*. The detailed explanations of the *Mobolic* components are given in Section 5.3.5.

Mobolic consists of five main components:

- **Static Analyzer (stAnalyzer)**. This component builds and analyses the Control Dependence Graph (CDG), and Data Dependence Graph (DDG) for an original Android app. Using the CDG and DDG graphs, it builds r-CDG for a mobile app. As an output, it returns the built r-CDG graph.
- **Application Instrumenter (appInster)**. Guided by the r-CDG graph, this component instruments the UID conditional statements in an app so that the corresponding UID symbolic constraints can be collected during the app execution. As an output, it returns an instrumented app.
- **Systematic Automated Test Driver (SAT)**. This component automatically and systematically explores an app GUI by interacting with UI widgets which are displayed on a mobile device screen. As an output, it returns a flag, which indicates whether the exercising procedure falls into a loop.
- **Input Scheduler (iScheduler)**. This component forms the UID symbolic path constraints leading to yet unexplored execution paths so that new app UIs can be discovered and exercised. As an output, it returns a symbolic path constraint (PC), which is formed for a particular editable UI widget.
- **Smart Customated Engine (smartCE)**. This component generates the relevant specific user inputs by finding user predefined, or solving UID symbolic PCs, or

Figure 5.3: Overview of Mobolic workflow

generating random inputs. As a result, app UIs which are constrained by such specific user inputs can be explored. As an output, it returns the relevant user inputs for the editable UI widgets.

Here, we overview the *Mobolic* workflow which is shown in Figure 5.3. *Mobolic* first runs *Static Analyzer* (*stAnalyzer*) to perform static analysis of the original Android app, and build the DDG, CDG, and r-CDG graphs. Guided by the r-CDG graph, *Application Instrumenter* (*appInster*) instruments the UID conditional statements. As a result, *appInster* re-builds an Android app with the instrumented UID conditional statements. *Systematic Automated Test Driver* (*SAT*) exercises the instrumented app by automatically and systematically exploring the app GUI. Also, during the exercising process, through an app instrumentation, *SAT* collects the UID symbolic constraints over the instrumented UID conditional statements along an execution path. From the collected UID symbolic constraints, *Input Scheduler* (*iScheduler*) forms the relevant UID symbolic path constraints (PCs) for the selected editable UI widgets, to explore yet uncovered execution paths. *Smart Customated Engine* (*smartCE*) generates relevant user inputs by solving the UID symbolic PCs with concrete values, generating user predefined, or random ones.

Next, we show an overall workflow of the *Mobolic* in Algorithm 5. Generally, *Mobolic* can be logically divided into three stages including (1) preparation (lines 2–6), (2) input generation (lines 9–20), and (3) exercising of an app under test (lines 21–34).

Stage#1. *Mobolic* performs static analysis and instrumentation of a target app, initializes variables, and runs the app. In line 2, *stAnalyzer* performs static analysis of the app and builds CDG and DDG with the aid of the Amandroid [186] tool. As an output,

Algorithm 5: Mobolic

Input: *app* – app under test

```
1 procedure Mobolic(app)
2   r-CDG = stAnalyzer(app);
3   instApp = appInster(app, r-CDG);
4   feui = < empty >;
5   f-UFG = < empty >;
6   Run(instApp);
7   while (TRUE) do
8     ui = GetCurrentUI();
9     foreach (editable widget ew ∈ ui) do
10      pc = iScheduler(ew);
11      in = smartCE(pc, ew);
12      InjectInput(in, ew);
13      if (pc.isFormed()) then
14        ldn = GetLastDecisionNode(pc);
15        if (ldn.isFullyCovered()) then
16          | pc.Remove(ldn);
17        end
18      end
19      UpdateStoredUIDConstraints(pc, ew);
20    end
21    infloop = SAT(f-UFG, ui);
22    lui = GetLastlyDiscoveredUI(f-UFG);
23    if (lui.isFullyExercised()) then
24      | feui.Add(lui);
25      | f-UFG.Remove(lui);
26      | if (lui.isCurrentUI()) then
27        | PressBack();
28      | end
29    end
30    if (infloop == TRUE) then
31      | ui = GetCurrentUI();
32      | lp = FindPathToLastlyDiscoveredUI(f-UFG, ui);
33      | ReplayPathToLastlyDiscoveredUI(lp);
34    end
35    if (f-UFG ∈ feui) then
36      | break while;
37    end
38  end
39 end
```

stAnalyzer returns (*r-CDG*). Guided by the *r-CDG* graph, in line 3, *appInster* creates an instrumented app (*instApp*) by replacing the original UID conditional statements in the original *app* with instrumented ones. In lines 4–5, *Mobolic* initializes a set of the fully exercised UIs (*feui*), and *f-UFG* as empty. The *f-UFG* model⁴ is used to guide the exercising procedure, and to ensure that *Mobolic* does not fall in an infinite loop, and *feui* is used for memorizing the fully exercised UIs. The *feui* and *f-UFG* are used together for a termination condition. *Mobolic* terminates the exercising procedure once all the discovered UIs are fully exercised. Next, *Mobolic* runs the instrumented app *instApp* (line 6).

It is important to note that *Mobolic* determines that an app UI is *fully exercised* only if (1) all its relevant non-editable widgets (if any) are fully exercised, and (2) all the decision nodes for each editable UI widget (if any) on an app UI are fully covered. In turn, *Mobolic* determines that a non-editable UI widget is *fully exercised* only if all its relevant UI events are executed. Also, *Mobolic* determines that a decision node for an editable UI widget is *fully covered* only if its corresponding UID statement with all its child sub-conditional statements are executed.

Stage#2. *Mobolic* performs an input generation. After analyzing, instrumenting, and running the instrumented app *instApp* in Stage#1, *Mobolic* first obtains the current UI (*ui*) displaying on an Android device (line 8). Next, in line 10, *iScheduler* forms a relevant symbolic PC for a selected editable UI widget (*ew*) by either appending newly collected UID symbolic constraints to an existing symbolic PC, or negating the last decision node of the existing symbolic PC. In line 11, *Mobolic* obtains a user input by executing *smartCE*, a customized input generation mechanism. First, *smartCE* searches for a user predefined input matching textual terms of the attributes “*text*”, “*resource-id*” and “*content-desc*” for a selected editable UI widget *ew*. If not found (e.g., input type is not recognized or does not exist), it searches for a default value which is assigned by the app itself. If the default value does not exist, it tries to solve the formed symbolic PC. In case if the formed symbolic PC (*pc*) cannot be solved (e.g., infeasible path), or *pc* is not formed yet (e.g., when *ew* is exercised for the first time), *smartCE* generates a random user input.

Mobolic injects the generated user input *in* into the selected editable UI widget *ew* (e.g., sets a text for the “*EditText*” widget) (line 12). In line 13, *Mobolic* checks if the symbolic PC *pc* has been formed. If so, *Mobolic* extracts the last decision node (*ldn*) from the formed symbolic PC *pc* (line 14). If *ldn* is fully covered (line 15), *Mobolic* removes it from *pc* (line 16). Next, for the selected *ew*, *Mobolic* updates the symbolic UID constraints by replacing the previously stored ones with constraints from the newly formed symbolic PC *pc* (line 19).

⁴In *Mobolic*, we use the same *f-UFG* model of an app GUI as in DroidDEV (see Chapter 4, Section 4.3.2 for details).

Stage#3. *Mobolic* exercises the app until all the discovered UIs in $f\text{-}UFG$ are fully exercised. After the input injection in Stage#2, *Mobolic* first obtains a flag (*infloop*), which indicates a loop occurrence in the exercising procedure, by executing *SAT* (line 21). For that purpose, *SAT* retrieves a non-editable unexercised relevant UI widget from the current UI *ui*. Afterwards, depending on the type of the widget, it generates the relevant UI-event to exercise the widget rendering a subsequent UI. Also, during exercising, *SAT* stores symbolic UID constraints for each editable widget which are encountered on the execution path. Next, *SAT* builds $f\text{-}UFG$ by adding the rendered subsequent UI with its widgets into $f\text{-}UFG$ only if the rendered subsequent UI does not exist in $f\text{-}UFG$ at this point. Note that exercising a non-editable widget could render a new subsequent UI. If so, *SAT* memorises such a link to store the transition from the exercised widget to its rendered subsequent UI. As such, *SAT* ensures that $f\text{-}UFG$ does not have any loops, and *Mobolic* will not exercise the app infinitely.

Mobolic obtains the lastly discovered UI (*lui*) from $f\text{-}UFG$ (line 22). The lastly discovered UI is a UI which is last added in $f\text{-}UFG$. Next, *Mobolic* checks if *lui* is fully exercised (line 23). If so, *Mobolic* adds *lui* into the set of the fully exercised UIs (*feui*) (line 24), removes it from $f\text{-}UFG$ and changes the lastly discovered UI in $f\text{-}UFG$ (line 25). In addition, if *lui* is the current UI (line 26), *Mobolic* generates a system key event “Back” (line 27), because *lui* has been fully exercised and removed from $f\text{-}UFG$. This event may return to any previously discovered app UI existing in $f\text{-}UFG$. Note that, depending on the app logic, system key event “Back” may exit the app. Such case is handled by *Mobolic* by restarting the app, and continuing the exercising procedure.

If *SAT* indicates that a loop has occurred in the exercising procedure, i.e., *infloop* is TRUE (line 30), *Mobolic* obtains the current UI *ui* (line 31), finds a path (*lp*) from *ui* to the lastly discovered UI in $f\text{-}UFG$ (line 32), and replays *lp* to reach the lastly discovered UI (line 33). As a result, at the next iteration, the lastly discovered UI will appear as the current UI *ui* displaying on a device screen (line 8), and the exercising procedure continues. The *FindPathToLastlyDiscoveredUI* function (line 32) performs the A* search by using the stored links in $f\text{-}UFG$ to find the shortest path leading to the lastly discovered UI in $f\text{-}UFG$. The *ReplayPathToLastlyDiscoveredUI* function (line 33) interacts with all the non-editable UI widgets, and generates user inputs for all the editable UI widgets along the path *lp* to reach the lastly discovered UI in $f\text{-}UFG$. Therefore, if there is any loop occurrence in the exercising procedure, *Mobolic* can easily track back the lastly discovered UI, and continue exercising yet unexercised UI widgets on the lastly discovered app UI. *Mobolic* automatically terminates the exercising process once all the discovered app UIs are fully exercised (i.e., the set of the fully exercised UIs *feui* contains all the app UIs from $f\text{-}UFG$) (lines 35–37).

5.3.5 Components

In this section, we describe in detail the functionality of the *Mobolic* components, and walk through their corresponding algorithms.

stAnalyzer: Static analyzer

To identify the UID statements, *stAnalyzer* performs a user-input dependence analysis on the CDG and DDG graphs [184, 185] of the app. In particular, *stAnalyzer* builds the graphs with the aid of a data flow analysis framework for Android apps, namely Amandroid [186]. Note that, however, Amandroid does not innately solve the problem of relating the Java code with the editable widgets on an app UI. To identify the UID statements, we implement our solution which leverages the intermediate representations Jawa⁵ and smali⁶ from Amandroid and Apktool,⁷ respectively.

The CDG graph is used to track intra- and inter-procedural conditional statements. From the CDG graph, *stAnalyzer* builds r-CDG to eliminate all the conditional statements which are irrelevant towards the reachability of a target UID conditional statement. The DDG graph is used to track the changes made to the UID statements which are data-dependent on user inputs. Therefore, all the changes can be reflected into the UID conditional statements and solved for symbolic user variables so that the corresponding concrete user inputs can be found.

In Algorithm 6, we show the main steps of how *stAnalyzer* performs static analysis of an app. First, *stAnalyzer* initializes the reduced Control Dependence Graph (*r-CDG*) as empty (line 2), and builds the Control Dependence Graph (*CDG*), and Data Dependence Graph (*DDG*) (lines 3–4). Next, guided by *DDG*, for each branch (*b*) in *CDG* (line 5), and for each decision node (*d*) in *b* (line 6), *stAnalyzer* checks if *d* has any data dependency on a user input (line 7). If so, *stAnalyzer* adds the selected branch *b* into *r-CDG* (line 8). The *stAnalyzer* terminates once all branches in the *CDG* graph are analyzed, and it returns *r-CDG* (line 13).

appInster: Application instrumenter

Guided by r-CDG, *appInster* performs an instrumentation of the UID conditional statements. Therefore, the instrumented UID conditional statements can be collected. Note that in the UID conditional statements, user-input-independent symbolic values will be replaced with the actual ones which are assigned by an app itself during run-time. The instrumented UID conditional statements may include “if-else”, “switch-case”, “for-” and “while-loop”.

⁵<http://pag.arguslab.org/jawa-language>

⁶<https://github.com/JesusFreke/smali>

⁷<http://ibotpeaches.github.io/Apktool>

Algorithm 6: stAnalyzer

Input: *app* – app under test
Output: *r-CDG* – reduced control dependence graph

```
1 procedure stAnalyzer(app)
2   r-CDG = < empty >;
3   CDG = BuildCDG(app);
4   DDG = BuildDDG(app);
5   foreach (branch b ∈ CDG) do
6     foreach (decision node d ∈ b) do
7       if (d.isDataDependentOnUserInput(DDG)) then
8         r-CDG.Add(b);
9         break;
10      end
11    end
12  end
13  return r-CDG;
14 end
```

There are several instrumentation techniques [187]. The *appInster* implements the most common one, i.e., bytecode instrumentation. Technically, *appInster* uses the smali intermediate language which represents *Dalvik* bytecode in a readable form. The *appInster* instruments the UID conditional statements by inserting additional probes (code) into smali which is obtained with the aid of Apktool, a tool for reverse engineering of Android apps. After instrumentation, *appInster* assembles an Android app from the instrumented smali code so that the app can be normally installed and executed.

In Algorithm 7, we show the main steps of how *appInster* instruments an app. Guided by the reduced Control Dependence Graph (*r-CDG*), for each branch (*b*) in *r-CDG* (line 2), and for each decision node (*d*) in *b* (line 3), *appInster* instruments *d* to obtain an instrumented decision node (*id*) (line 4). Afterwards, it replaces *d* with *id* in the original (*app*), and generates an instrumented app (*instApp*) (line 5). The *appInster* terminates once all branches in *r-CDG* are processed, and returns the instrumented app *instApp* (line 8).

SAT: Systematic automated test driver

SAT is primarily designed for performing a black box-style functional automated UI testing, where a test code does not rely on internal implementation details of a target app. *SAT* implies an event-driven automated UI test driver. It automatically and systematically explores an app GUI by interacting with non-editable UI widgets which are displayed on an Android device screen. In particular, *SAT* exercises all the relevant UI widgets one-by-one, following a sequence in which widgets are innately located in the UI hierarchy view, from up to bottom, and from left to right. To qualify for a systematic GUI exploration, *SAT*

Algorithm 7: appInster

Input: *app* – original app, *r-CDG* – reduced control dependent graph

Output: *instApp* – instrumented app

```
1 procedure appInster(app, r-CDG)
2   foreach (branch b  $\in$  r-CDG) do
3     foreach (decision node d  $\in$  b) do
4       id = InstumentDecisionNode(d);
5       instApp = ReplaceDecisionNode(d, id, app);
6     end
7   end
8   return instApp;
9 end
```

must ensure that an exercising process is not interrupted. The systematic GUI exploration is commonly interrupted by loops existing in a GUI flow. Thus, to address this issue and ensure that *SAT* performs the systematic GUI exploration, *Mobolic* implements the A* search algorithm. For example, if a loop occurs during exercising, in order to return to the app UI on which the exercising process has been interrupted, *Mobolic* performs the A* search in *f-UFG* to find the shortest path from the current app UI, displaying on an Android device screen, to the lastly discovered app UI. In particular, once a loop has occurred, *SAT* indicates an interruption in the exercising process, i.e., it returns TRUE to the exercising procedure, by which it signals *Mobolic* to perform the A* search. As a result, guided by A*, *Mobolic* returns the lastly discovered app UI so that *SAT* can continue the GUI exploration from that lastly discovered app UI.

During exercising, *SAT* automatically (1) builds the *f-UFG* model which is used to continuously guide the exercising procedure without falling in a loop, and (2) collects the UID symbolic constraints along the execution path through the app instrumentation. Note that *SAT* collects the UID symbolic constraints over the UID conditional statements once their corresponding UID conditional statements are encountered (i.e., their code is executed). For the comprehensive exercising of an app GUI, *Mobolic* supports the most common UI user actions which are listed in Table 5.1.

Table 5.1: Types of events supported by *Mobolic SAT*

Event origin:type	Actions	Hardware keys
UI:Touch	Click; LongClick	–
UI:Motion	Swipe; Scroll; Pinch	–
UI:Trackball	Roll; Press	–
UI:Keypress	Input Injection	–
System:“Major” navigation	–	“Home”; “Back”; “Menu”

In Algorithm 8, we show the main steps of how *SAT* guides the exercising procedure by building *f-UFG* and collecting the UID symbolic constraints. *SAT* first sets the flag

of a loop occurrence (*inloop*) to be TRUE (line 2), and adds the current UI (*ui*) with its widgets into *f-UFG* (line 4) if *ui* is not yet in *f-UFG* (line 3). In line 6, *SAT* obtains the relevant unexercised non-editable UI widget (*nw*). Next, *SAT* exercises the relevant widget by generating the relevant UI event for the widget *nw* (line 7). Note that to generate the relevant UI event, *SAT* analyzes the attributes of the widget *nw*. Firing the UI event upon *nw* executes a certain code in the app. As such, for each editable widget *ew* on the current UI *ui*, *SAT* stores all the UID symbolic constraints encountered on the execution path (lines 8–10). By firing an UI event upon the widget, the current UI could be changed. As such, the current UI *ui* must be updated (line 11). If the current UI *ui* is not in *f-UFG* (line 12) (i.e., *ui* is a newly discovered UI), *SAT* adds *ui* with its widgets into *f-UFG* (line 13), and adds the link (*nw, ui*) into *f-UFG* (line 14) to memorize the transition from *nw* to *ui*. In addition, *SAT* sets the flag *inloop* to FALSE (line 15). In line 17, *SAT* returns the *inloop* flag to *Mobolic*. If the returned flag *inloop* is FALSE, it signals *Mobolic* that there is no loop occurrence in the exercising procedure, and thus *SAT* can continue with the newly discovered UI. Otherwise, if the returned flag *inloop* is TRUE, it signals *Mobolic* that there is a loop occurrence in the exercising procedure, and thus *SAT* requires *Mobolic* to find the lastly discovered UI to continue the exercising procedure.

Algorithm 8: SAT

Input: *f-UFG* – finite state GUI flow graph, *ui* – current UI

Output: *inloop* – flag of loop occurrence

```

1 procedure SAT(f-UFG, ui)
2   inloop = TRUE;
3   if (ui  $\notin$  f-UFG) then
4     | f-UFG.Add(ui);
5   end
6   nw = GetUnExercisedRelevantWidget(ui);
7   ExerciseNonEditableWidget(nw);
8   foreach (editable widget ew  $\in$  ui) do
9     | StoreEncounteredUIDConstraints(ew);
10  end
11  ui = GetCurrentUI();
12  if (ui  $\notin$  f-UFG) then
13    | f-UFG.Add(ui);
14    | f-UFG.AddLink(nw, ui);
15    | inloop = FALSE;
16  end
17  return inloop;
18 end

```

iScheduler: Input scheduler

This component forms the UID symbolic PCs based on the UID symbolic constraints collected by *SAT*. The *iScheduler* ensures that the relevant UID symbolic PCs are formed for every editable UI widget to cover as many execution paths as possible. *Mobolic* implements an iterative process, and for every iteration, the newly generated user inputs may lead to unexplored program paths with different app functionality. As such, to explore as many execution paths as possible, *iScheduler* derives a UID symbolic PC at each iteration. In particular, *iScheduler* forms a UID symbolic PC by tweaking the collected UID symbolic constraints as follows:

1. For a “*if*”-statement, its corresponding UID symbolic constraint is constructed. The constructed UID symbolic constraint is then appended to the earlier formed UID symbolic PC of the nearest ancestor UID conditional statement to form the symbolic PC of the “*if*”-statement. Note that if the ancestor UID conditional statement does not exist for a particular “*if*”-statement, the UID symbolic PC will only consist of the UID symbolic constraint corresponding to its “*if*”-statement.
2. For a “*else*”-statement, its UID symbolic constraint is constructed from the corresponding “*if*”-statement by applying the logical negation operator “NOT”. The constructed UID symbolic constraint is then appended to the earlier formed UID symbolic PC of the nearest ancestor UID conditional statement to form the symbolic PC of the “*else*”-statement.

In practice, for all but trivial apps, the user inputs may be required on any UI of an app. Therefore, the user inputs commonly depend on each other. Considering such dependencies, *iScheduler* uses the following rules to eliminate missing any yet undiscovered app UIs:

1. For each editable UI widget, at a time, *iScheduler* forms only one relevant symbolic PC since each editable UI widget may have several distinct symbolic PCs depending on an app logic.
2. For each editable UI widget, at a time, *iScheduler* forms the symbolic PC whose solution (i.e., a generated user input) may contribute to discovering yet unexplored app UIs.

In Algorithm 9, we show the main steps of how *iScheduler* forms a relevant symbolic PC for a selected editable UI widget. First, *iScheduler* sets the symbolic PC (*pc*) to empty (line 2), and obtains all the UID symbolic constraints (*sc*) stored by *SAT* for the selected editable UI widget (*ew*) (line 3). Next, *iScheduler* checks if the obtained UID symbolic constraints *sc* are not empty (line 4). If so, from *sc*, *iScheduler* forms a symbolic PC

pc (line 5). In case if sc does not have any new symbolic UID constraints (i.e., the constraints which are not included into the returned pc at the previous iterations) (line 6), $iScheduler$ negates the last decision node in the formed symbolic PC pc (line 7). As a result, $iScheduler$ forms and returns the relevant symbolic PC whose solution (i.e., a generated user input) may lead to discovering yet unexplored app UIs (line 10).

smartCE: Smart customized engine

The *smartCE* is a customized engine which is used to automatically generate specific user inputs. The *smartCE* implements our novel customized input generation mechanism which handles concrete (via solving symbolic PCs which are collected through the app instrumentation), user predefined (via exploiting particular textual attributes of the editable UI widgets), and random (numerical or textual) inputs.⁸

For the concrete input generation, *smartCE* takes the UID symbolic PC formed by *iScheduler*, and solves it to obtain a concrete user input. Note that since symbolic execution is naturally time-consuming, the found user input values are stored for every editable widget, and could be further re-used in the following exercising to shorten the exercising time by not solving the same UID symbolic PCs multiple times. Solving a UID symbolic PC, *smartCE* finds concrete user inputs which satisfy a particular symbolic PC corresponding to the execution path in the app code. Therefore, concrete user inputs allow *Mobolic* to discover app UIs which are constrained by such the user inputs. In case if a formed symbolic PC cannot be solved (e.g., infeasible path), or it is not formed yet (e.g., when an editable widget is exercised at the first time), *smartCE* will generate random user input. The random user input ensures that the exercising procedure proceeds without any interruptions.

Algorithm 9: iScheduler

Input: ew – editable widget
Output: pc – symbolic path constraint

```

1 procedure iScheduler( $ew$ )
2    $pc = \langle empty \rangle$ ;
3    $sc = \text{GetStoredUIDConstraints}(ew)$ ;
4   if ( $sc.isEmpty()$ ) then
5      $pc = \text{FormSymbolicPathConstrain}(sc)$ ;
6     if ( $sc.hasNotNewSymbolicConstraints()$ ) then
7       NegateLastDecisionNode( $pc$ );
8     end
9   end
10  return  $pc$ ;
11 end

```

⁸For the user predefined and random user inputs, see Section 4

The *smartCE* engine is implemented in *Java*, and uses the Symbolic Math Toolbox in MATLAB [188]. The Symbolic Math Toolbox provides a large set of functions for solving and manipulating symbolic math expressions. The symbolic math expressions can be solved either analytically or using variable precision arithmetic. The *smartCE* communicates with the MATLAB symbolic engine by using the third-party *Java* library, namely *matlabcontrol*.⁹

In Algorithm 10, we show the main steps of how *smartCE* generates a user input for an editable UI widget. First, *smartCE* sets the generated input (*in*) to empty (line 2). Next, *smartCE* extracts the relevant input types from the UI widget attributes such as “*text*”, “*resource-id*” and “*content-desc*”. It identifies the relevant input type for the editable UI widget *ew* (line 3) by scanning through DBT2,¹⁰ which consists of the commonly used input types, and their corresponding default or user predefined input values. Based on the found input type *iType*, *smartCE* obtains the corresponding input value from DBT2 (line 5). If the input type is not found in DBT2, *smartCE* applies an input generation procedure (lines 7–15). If the formed *pc* is not empty (line 7), *smartCE* tries to generate a concrete user input *in* by solving the given *pc* (line 8). If no solution is found for the *pc* (line 9), *smartCE* tries to obtain a default value (e.g., an actual value or hint) which is, by default, set by the app itself (line 10). If the default value does not exist (line 11),

Algorithm 10: smartCE

Input: *pc* – symbolic path constraint, *ew* – current editable widget

Output: *in* – generated input for current editable widget

```

1 procedure smartCE(pc, ew)
2   in = < empty >;
3   iType = DBT2.GetType(ew);
4   if (iType.isFound()) then
5     in = DBT2.GetCustomValue(iType);
6   else
7     if (pc.isNotEmpty()) then
8       in = GenerateConcreteInput(pc);
9       if (in.isEmpty()) then
10        in = ew.GetDefaultValue();
11        if (in.isEmpty()) then
12          in = GenerateRandomInput();
13        end
14      end
15    end
16  end
17  return in;
18 end

```

⁹<https://www.cs.virginia.edu/~whitehouse/matlab/JavaMatlab.html>

¹⁰see Section 4.4.6 in Chapter 4

smartCE generates a random user input *in* (line 12). In the end, *smartCE* returns the generated user input *in* (line 17).

5.4 Design and Implementation of Mobolic: Practical Experience

In this section, we provide the details of the *Mobolic* practical experience which are learned during its design and development phases. The main question concerns about an automated generation of the relevant user inputs via the **customated input generation** using concrete user values. We list the questions below as follows:

1. How do we relate editable UI widgets with the Java code?
2. How do we implement concolic execution using MATLAB symbolic engine?
3. How do we generate concrete textual user inputs?

5.4.1 How do we relate editable UI widgets with the Java code?

To use the generated concrete user inputs, *Mobolic* needs to relate the editable UI widgets to a level of the Java code where they are used. Having such relations, *Mobolic* knows in which particular editable UI widget the concrete user input should be injected. To resolve the problem of relating the Java code with the user interface, *Mobolic* leverages the intermediate representations (IRs) Jawa,¹¹ and smali which are generated by Amandroid, and Apktool, respectively. For example, we have a dice game *Yahtzee*. On its first UI, we need to indicate a number of rounds to play. Using the UI hierarchy viewer,¹² a tool from the Android SDK for GUI inspection, we can see that the editable UI widget relating to a number of rounds has a *resource-id* attribute with the value *com.tum.yahtzee:id/editText_rounds*, where *com.tum.yahtzee* is the app package name, and *editText_rounds* is the *id* of the UI widget. In fact, every Android app contains *ids* for all its UI widgets. Therefore, when *Mobolic* disassembles the app into smali using Apktool, it retrieves *ids* in a human-readable form in the file *R\$id.smali*. From the *resource-id* attribute, *Mobolic* uses the package name *com.tum.yahtzee* to construct the path “*../com/tum/yahtzee/*” where the file *R\$id.smali* is located on disk. Next, *Mobolic* scans through *R\$id.smali*, and searches for *editText_rounds*. In particular, *Mobolic* performs search with a text which matches the string “*editText_rounds:I =* ”. Once it has found the matching string, *Mobolic* retrieves an integer value which is represented in the hexadecimal system on the right of the symbol ‘=’, and corresponds to *editText_rounds*,

¹¹<http://pag.arguslab.org/jawa-language>

¹²<https://developer.android.com/studio/profile/hierarchy-viewer>

e.g., 0x7f050010. Next, *Mobolic* uses Amandroid to disassemble the app, and obtain Jawa. Note that in Jawa, the UI widgets *ids* are represented in the decimal system, so *Mobolic* converts 0x7f050010 into the decimal value, i.e., 2131034128. To find a name of the Java object with *id* 2131034128 which corresponds to the widget on an app UI, *Mobolic* scans through all the *.jawa* files, and searches for the Jawa object with *id* 2131034128. Once the Jawa object is found, *Mobolic* searches for the nearest *AssignmentStatement*, and by using the data dependencies, it identifies that the Jawa object with *id* 2131034128 has the corresponding name *roundsText* in the Java code. As a result, by using the *ids* from smali and Jawa, *Mobolic* identifies that the Java object with the name *roundsText* relates to the editable UI widget *editText_rounds*.

5.4.2 How do we implement concolic execution using MATLAB symbolic engine?

Mobolic implements a concolic execution technique using a MATLAB symbolic engine to solve symbolic path constraints for user values in the UID conditional statements. In summary, the concrete input generation mechanism works as follows. During an app execution, on the current app UI, *Mobolic* collects symbolic constraints where user inputs are involved. Next, it forms the symbolic path constraints and writes them to the file “*solution.m*” which can be executed by MATLAB. Using the ‘*matlabcontrol*’ library, *Mobolic* sends the “*solution.m*” file to the MATLAB symbolic engine. After solving the symbolic path constraints, the solutions (if several) for the symbolic variables will be written into the “*solution.res*” file by MATLAB. The found solutions are the concrete values which will be read and further managed by *Mobolic* by injecting them into the respective editable UI widgets on the current app UI.

Giving the summary as above, we next explain in detail how the concrete user inputs are generated. For that purpose, we use an architectural scheme of the customized input generation mechanism and example test app, which are shown in Figures 5.4 and 5.5, respectively. In Figure 5.5, we show the test app with a single app UI which contains two editable UI widgets, and one button. We denote the editable UI widgets as EW_1 and EW_2 , and the button as B_1 . We assume that the test app has a portion of the code as shown in Figure 5.6, where the symbolic variables x and y associate with the editable UI widgets EW_1 and EW_2 , respectively. That is, a user value from EW_1 will be assigned to x , and a user value from EW_2 will be assigned to y during the code execution.

In Figure 5.4, we show an overall architecture of the customized input generation mechanism. Before *Mobolic* starts the testing process, it needs to perform app instrumentation so that it can collect the UID conditional statements during the app execution. Using *r-CDG*, *Mobolic* instruments the UID conditional statements of the test app as follows. It inserts an additional code (probe) directly into the original UID conditional

Figure 5.4: Architectural design of customized input generation mechanism

Figure 5.5: Example mobile app UI

```

1: . . . // some Java code
2: EditText efInput1 = (EditText) findViewById(R.id.editField1) ;
3: EditText efInput2 = (EditText) findViewById(R.id.editField2) ;
4: int x = Integer.parseInt(efInput1.getText().toString()) ;
5: int y = Integer.parseInt(efInput2.getText().toString()) ;
6: . . . // some other Java code
7: if ((x > 6)&&(y <= -9)&&(x + y == C)) {
8:     /* do something */
9: }
10: else {
11:     /* do something else */
12: }

```

Figure 5.6: Example code of mobile app with UID conditional statement

statements, and places the probe as the first (leftmost) condition of the “if” statement to ensure that it is always executed since (1) the conditions in the “if” statement are always executed before it jumps to its “else” branch, and (2) the conditions in the “if” statement are always executed from left to right. So, our probe is always executed regardless of whether all the conditions in the “if” statement are satisfied or not. In turn, our probe is represented as a function, namely *getUID*, which always returns the boolean TRUE to its “if” statement. As such, we ensure that our probes do not corrupt the original app logic during instrumentation. The function *getUID* has a single input argument which is a string representing a UID conditional statement. This function writes the obtained UID statement into a file, namely “*buffer.sym*”. The “*buffer.sym*” file is used by *Mobolic* to obtain the collected UID statements for the concrete input generation.

After the instrumentation process is completed, the code with our probe will be as shown in Figure 5.7, and *Mobolic* starts the testing process using the instrumented app. (i) *Mobolic* runs the app, generates two random user inputs for the editable UI widgets EW_1 and EW_2 , and clicks the button B_1 . We note that since EW_1 and EW_2 are exercised at the first time, *Mobolic* can only generate random inputs, e.g., 11 and 8. After clicking B_1 , the conditions in the instrumented “if” statement (see Figure 5.7) will be executed. Therefore, regardless of the generated inputs (it is random inputs at this moment), the *getUID* function will also be executed. (ii) So, it obtains the UID statement “0 : (x > 6)&&(y <= -9)&&(x + y == C) : *editField1*, *editField2* : x, y : 10”, and writes it into the “*buffer.sym*” file.

An instrumented UID conditional statement is formed according to the pattern as follows: “*id:cs:edit_ids:syms:consts*”, where ‘*id*’ is a unique number of the UID statement (the number is assigned during the instrumentation process), ‘*cs*’ is a conditional UID statement itself, ‘*edit_ids*’ are unique names (can also be one name) of the editable UI widgets, which are given by a developer (these names can be found in the values of the *resource-id* XML-attributes [see Section 5.4.1] for the editable UI widgets when we

obtain a view hierarchy of an app UI), ‘*syms*’ are symbolic variables (can also be one variable) of the UID statement (the values of these variables are manipulated by a user through editable UI widgets), and ‘*consts*’ are concrete (runtime) values (can also be one value) of the symbolic constants (they will replace their respective symbolic constants in *cs* in the order from left to right) in the UID statement. It is important to note that the sequences of elements in the sets of ‘*edit_ids*’ and ‘*syms*’ must follow the order in which the symbolic variables correspond to their respective editable UI widgets. If there are more than one editable UI widget, symbolic variable and/or symbolic constant in a single UID statement, they will be separated by the comma (,) symbol.

```

1: . . . // some Java code
2: EditText efInput1 = (EditText)findViewById(R.id.editField1);
3: EditText efInput2 = (EditText)findViewById(R.id.editField2);
4: int x = Integer.parseInt(efInput1.getText().toString());
5: int y = Integer.parseInt(efInput2.getText().toString());
6: . . . // some other Java code
7: if (getUID("0:(x > 6)&&(y <= -9)&&(x + y == C):editField1,editField2:x,y:"+C)
8:   && ((x > 6)&&(y <= -9)&&(x + y == C))) {
9:   /* do something */
10: }
11: else {
12:   /* do something else */
13: }

```

Figure 5.7: Example code of mobile app with instrumented UID conditional statement

In our case, $id = 0$, $cs = (x > 6) \&\& (y \leq -9) \&\& (x + y == C)$, $edit_ids = editField1, editField2$, $syms = x, y$, and $consts = 10$. For ‘*consts*’, we use ‘10’ as an example; however, it can be any concrete value assigned during runtime. At this moment, the UID statement in “*buffer.sym*”. (iii) Next, *Mobolic* reads the statement from the file (if there are several statements, it reads all of them at once, and then erases the file content; next, it processes the statements one-by-one) and erases the file content. Once the statement has been read from the “*buffer.sym*” file, *Mobolic* stores it in the respective instances of the editable UI widgets of the current app UI using their corresponding names from ‘*edit_ids*’. *Mobolic* uses a stored UID statement for the following purposes: (1) to identify if a statement has previously been solved by comparing its ‘*id*’ with a new statement from the “*buffer.sym*” file, (2) to generate another concrete inputs by negating constraints in a statement to execute “*else*” branch of the statement, and (3) to identify editable UI widgets to where the generated inputs should be injected by using the symbolic values from ‘*syms*’ which are associated with their respective editable UI widgets through the names in ‘*edit_ids*’. Afterwards, *Mobolic* uses the runtime values from ‘*consts*’ to replace the symbolic constants in ‘*cs*’ in the order from left to right. In our case, the runtime value ‘10’ will replace the symbolic constant ‘*C*’, so $cs = (x > 6) \&\& (y \leq -9) \&\& (x + y == 10)$.

(iv) Next, *Mobolic* prepares and sends the “*solution.m*” MATLAB file where the symbolic constraints of the UID conditional statement will be formed in a system of

algebraic equations and/or inequalities if there are more than one symbolic constraint in the UID statement. An example MATLAB code is shown in Figure 5.8. In *line 2*, we declare symbolic variables x and y . In *lines 3, 4*, we add symbolic assumptions for x and y , respectively. For the symbolic assumptions, we use the symbolic constraints which involve only one symbolic variable, e.g., $(x > 6)$ or $(y \leq -9)$. For each symbolic assumption, we explicitly add one extra condition to simplify the search of solutions. In particular, we replace the symbol ' $>$ ' with ' $== 1+$ ', and the symbol ' $<$ ' with ' $== -1+$ ', so that from $(x > 6)$ and $(y \leq -9)$, we form two extra conditions with equalities $(x == 1+6)$ and $(y == -1+-9)$, which are separated from them by logical 'OR', the symbol '|' in MATLAB (see *lines 3, 4* in Figure 5.8). In addition, if a symbolic constraint has the symbol ' $\leq$ ' or ' $\geq$ ', we split the constraint into two symbolic assumptions, e.g., $(y \leq -9)$ will be replaced with $(y == -9) | (y < -9)$ (see *line 4* in Figure 5.8). In *line 5*, we declare our main symbolic condition $(x + y == 10)$. For the main symbolic condition, we use a symbolic constraint which involves all the symbolic variables from the UID conditional statement. In *lines 8–17*, we try to solve the main symbolic condition with the symbolic variables assumptions for different solution domains including 'integer', 'real' and without any domain specified. Next, we write found solutions into the "*solution.res*" file. If solutions are not found, the "*solution.res*" file will contain only the symbolic variables names without their concrete values; in this case, *Mobolic* will assign random values for the symbolic variables.

```

1: for iteration = 1:9 %solve (x>6)&&(y<=-9)&&(x+y==10) — UID conditional statement
2:     syms x y %symbolic variables
3:     assumeAlso(( x==1+6 ) | ( x>6 )) %symbolic assumption for 'x'
4:     assumeAlso(( y== -1+-9 ) | ( y== -9 ) | ( y<-9 )) %symbolic assumption for 'y'
5:     cond = ( x+y==10 ); %main symbolic condition
6:     %search solutions in 'integer', 'real' and without domain assumptions
7:     try
8:         if 1 == iteration
9:             assumeAlso( x,'integer' )
10:            assumeAlso( y,'integer' )
11:            [ x,y ] = solve( cond,[ x,y ],'PrincipalValue',true );
12:        elseif 2 == iteration
13:            assumeAlso( x,'integer' )
14:            assumeAlso( y,'integer' )
15:            [ x,y ] = solve( cond,[ x,y ],'Real',true );
16:            . . . %do next iterations (until iteration=9) for other solution domains
17:        end
18:    catch e
19:        continue %try another solver
20:    end
21:    if ~isempty(x)
22:        . . . %write solutions 'x' and 'y' into the 'solution.res' file
23:        return %exit the procedure since solutions are found
24:    end
25:    %go to the next iteration
26: end

```

Figure 5.8: Example MATLAB code for solving UID conditional statement

⑤ After solving the symbolic constraints for x and y symbolic variables, MATLAB

writes the found solutions in the “*solution.res*” file according to the pattern as follows: “*sym=val*”, where *sym* is a symbolic variable for which a symbolic path constraint is solved, and *val* is a solution (concrete value) which the symbolic variable can take. In our case, the “*solution.res*” file will contain two lines with the following data, e.g., $x = 19$ and $y = -9$ (solutions are found at ‘1 == iteration’, see line 8 in Figure 5.8). (vi) Next, *Mobolic* reads the file and stores the concrete values in the respective instances of the editable UI widgets which are associated with their symbolic variables. Once the file has been read, *Mobolic* erases the file content. (vii) Afterwards, *Mobolic* injects the concrete values into their editable UI widgets on the current app UI and clicks B_1 . With these values, ‘19’ and ‘-9’ for EW_1 and EW_2 , respectively, all the constraints in the “if” condition of the UID statement will be satisfied, so the code in the “if” branch will be executed and thus tested. At the next iteration of the testing process, *Mobolic* negates the main symbolic constraint in the stored UID statement to cover “else” branch. It prepares another “*solution.m*” file and sends it to MATLAB for solving. An example MATLAB code is the same as in Figure 5.8 except for lines 2, 5 where the main symbolic condition is replaced with the negated one $\sim (x + y == 10)$ where the symbol ‘ $\sim$ ’ is logical ‘NOT’ in MATLAB. After solving the symbolic constraints for x and y , *Mobolic* reads the “*solution.res*” file which contains two lines with the following data, e.g., $x = 7$ and $y = -9$ (solutions are found at ‘2 == iteration’, see line 12 in Figure 5.8). Using these concrete values for EW_1 and EW_2 , all the constraints in the “if” condition of the UID statement will not be satisfied, so the “else” branch will be executed and thus tested.

5.4.3 How do we generate concrete textual user inputs?

The **customated input generation** mechanism encompasses three types of the generated user inputs: *Concrete*,¹³ *Random/Default* and *UI-context-aware*.¹⁴ For each input generation method, we set a priority. So, *UI-context-aware* is the first method to perform, *Concrete* is the second method to perform, and *Random/Default* is the last method to perform. Note that once any of the methods returns a generated input, *Mobolic* stops the input generation process, and uses the returned input in the exercising process. Defining such the sequence, *Mobolic* finds the most relevant textual inputs to the current app UI. In the following sections, we describe each of the methods, and explain how they are implemented in practice.

¹³The *Concrete* input generation is newly introduced in *Mobolic*.

¹⁴The *Random/Default*, and *UI-context-aware* input generation is implemented in DroidDEV (see Chapter 4, Section 4.4), and exploited by *Mobolic*.

Concrete textual user input generation

In *Mobolic*, we use the idea of dynamic symbolic execution (concolic execution). That is, in a symbolic constraint, we assume that all the user-input dependent variables (i.e., which a user can inject via editable UI widgets on an app screen) are symbolic, while all the other variables are concrete which are computed during runtime. For example, we collect the symbolic constraint $x > C$, where x is a symbolic variable (can take any user value), and C is a symbolic constant which is assigned during runtime, e.g., $C = 5$. So, our collected symbolic constraint will be changed to the explicit form such as $x > 5$.

Using concolic execution, we implement the *Concrete* input generation method. It enables *Mobolic* to generate inputs which should be an exact value or within exact range. For example, if an app requires the 'age' input, and the input is validated in the code, e.g., $(age > 0) \ \&\& \ (age < 16)$. For such scenario, the *Concrete* input generation method will generate a input value to satisfy the 'age' condition. To extract $(age > 0) \ \&\& \ (age < 16)$, *Mobolic* performs the app code instrumentation by running its *appInster* component. The app code instrumentation is used to collect the UID conditional statements. *Mobolic* uses these conditional statements to form the UID symbolic PCs. In turn, the UID symbolic PCs are to be solved by *Mobolic's* component *smartCE* which is built on the top of the MATLAB symbolic engine. So, in particular, when *smartCE* solves the UID symbolic PC $\{(age > 0) \ \&\& \ (age < 16)\}$, it returns a numeric value within the expected range $(0, 16)$, which will be defined by an underlying symbolic engine from MATLAB.

To summarise, we stress that *Mobolic* generates concrete user inputs using the dynamic symbolic execution technique. It collects symbolic constraints via the app code instrumentation and forms possible symbolic path constraints. *Mobolic* symbolically executes only the UID conditional statements while other statements are executed concretely. That is, at runtime, the other variables in the UID conditional statements take concrete runtime values while symbolic values are to be found via solving their corresponding symbolic constraints.

Our implementation of the *Concrete* input generation method is different from the conventional one. It differs in a way that we execute symbolically only the UID conditional statements, while the conventional method attempts to symbolically execute all possible conditional statements in the code. Using the *Concrete* input generation method, *Mobolic* aims to cover all the UID conditional statements it has encountered on the execution path. For that purpose, *Mobolic* forms symbolic path constraints for every discovered editable widget on the app UI and solves it using the MATLAB symbolic engine. In practice, along with the recorded symbolic path constraints, *Mobolic* stores the generated concrete inputs so that they can be re-used for the next exercising cycle. It is important to note that *Mobolic* continues generating the same user inputs for the corresponding editable widgets on the required app UIs until all their subsequent app UIs,

depending on these inputs, are fully exercised.

Mobolic interacts with MATLAB using ‘*matlabcontrol*’ library. First, *Mobolic* prepares a MATLAB script. It consists of the algebraic equations and/or inequalities depending on the symbolic expressions in the formed symbolic path constraint. Next, *Mobolic* writes a prepared script into the “*solution.m*” file. Note that the formed symbolic path includes symbolic constraints concatenated with the logical “&&” operator so that, in MATLAB, the symbolic path constraints can be represented as a system of algebraic equations and/or inequalities. As such, it guarantees that the system of algebraic equations and/or inequalities must be satisfied in order to have a satisfactory solution for the whole symbolic path constraint. Next, *Mobolic* uses ‘*matlabcontrol*’ to send the script to the MATLAB symbolic engine for execution. Once the system is solved for every symbolic variable, MATLAB returns the solution by writing it into the “*solution.res*” file. It records the found values for every symbolic variable in the symbolic path constraint. Note that the solution can be returned in an integer form (e.g., $-1, 0, 5$ and others), or fractional form (e.g., $\frac{3}{7}, \frac{2}{5}$ and others). The fractional form values will be injected into an editable widget in decimal values with floating point, e.g., $\frac{3}{7}$ as 0.42857142857, or $\frac{2}{5}$ as 0.4.

Mobolic uses MATLAB to solve the symbolic paths constraints, solutions of which lie in the numerical domain. However, there are cases when a textual solution is needed. For that purpose, we implemented a string resolver (in Java) to support three main text operations: “*equals*”, “*compare*”, and “*contains*”. So, in order to find a textual input, we parse the conditional expression into two parts and obtain a constant string value from either side. For example, “*name.equals(s)*”, or *s.equals(“name”)* are equivalent statements, where “*name*” is a string constant, and “*s*” is a symbolic variable which belongs to the UID conditional statement. In order to find a value of “*s*”, we extract the constant value from another side of “*s*”, i.e., from the most left, or the most right position of the statement. For example, in the code, it can be written like *s.toString().toLowerCase().equals(“name”)*, or “*name.equals(s.toString().toLowerCase())*”.

Our string resolver finds textual inputs assuming that “*equals*”, “*compare*”, and “*contains*” answer the following questions: for “*equals*”: is it equal to/same as ? – ‘Yes’, for “*compare*”: is it more/less compare to ? – ‘Yes’, for “*contains*”: does it contain ? – ‘Yes’. So, if an actual condition in Java is written with a meaning, e.g., ‘NOT contain’ or ‘NOT equal’ (!“*name.contains(s)*” or !“*name.equals(s)*”), to satisfy the conditions, our string resolver will negate the found solution for ‘Yes’ by providing a reversed string if ‘Yes’ solution is not empty, or a random non-empty string if ‘Yes’ solution is an empty string, thus ensuring that ‘*if*’ and ‘*else*’ logical branches are covered.

5.5 Mobolic: Demo

In this section, we show a workflow of *Mobolic* on the app whose UFG is shown in Figure 5.9, Section 5.3.2. In this demo, we show how *Mobolic* explores the app GUI using the *f-UFG* model, and how *Mobolic* generates user inputs via the customized mechanism. Note that, in this demo, we focus on a symbolic part of the customized input generation mechanism due to the complexity of demonstrating a mixture of all possibly generated inputs. For more details about *UI-context-aware* and *Random/Default* user inputs generation, refer to our conference paper [101], and Section 4.4 in Chapter 4.

Figure 5.9: Example UFG of mobile app

In our demo, we assume that the app consists of three UIs. On u_1 , the app has the editable widget e_{11} . The injected input for e_{11} is passed as an input parameter to the function “*foo*” whose code, and *r-CDG* are shown in Figures 5.1 and 5.2, Section 5.3.2, respectively. Note that, in Figure 5.1, *code#4* in line 15 does not depend on the user input. As such, it may be executed or not depending on the system time which is obtained at that moment when the function “*foo*” is invoked, and the conditional statement in line 14 is executed. The function “*foo*” is always invoked when a relevant UI event is fired upon the non-editable widget w_{12} . Without loss of generality, we assume that the app code does not have any other UID conditional statements except those in the function “*foo*”.

Step#1. First, *Mobolic* executes *stAnalyzer* which builds *r-CDG* as shown in Figure 5.2. From Figure 5.2, we can see that the function “*foo*” has three execution paths $1 \Rightarrow 2$, $1 \Rightarrow 4 \Rightarrow 5$ and $1 \Rightarrow 4 \Rightarrow 6$, and two decision nodes ① and ④, which are constrained by the user input from the editable widget e_{11} . Next, guided by *r-CDG*, *Mobolic* executes *appInster* to instrument the app. *Mobolic* sets a set of the fully exercised UIs *feui* and *f-UFG* to be empty.

Step#2. Next, *Mobolic* runs the instrumented app, and obtains the first UI u_1 . Next, *Mobolic* executes *iScheduler* and *smartCE* to generate a relevant user input. Since the editable widget e_{11} is exercised for the very first time, *smartCE* generates a random user input, e.g., 11. *Mobolic* injects the input into e_{11} .

Step#3. Afterwards, *Mobolic* executes *SAT* to exercise the non-editable widgets, build *f-UGF*, and collect the UID symbolic constraints through the app instrumentation. *SAT* adds u_1 with its widgets into *f-UGF*, and finds the first unexercised relevant non-editable widget on u_1 , i.e., w_{12} . *SAT* exercises w_{12} by generating the relevant UI-event for w_{12} , e.g., “Click” event. Upon exercising w_{12} , *Mobolic* invokes the function “foo” so that *code#1* is executed since the UID conditional statement $(x > 5)$ corresponding to the decision node ① is satisfied by the random input, i.e., 11, which has been generated in Step#2. Also, after clicking on w_{12} , u_2 is discovered. *SAT* stores the corresponding symbolic constraint $[(x > 5)]$ which is encountered on the execution path $1 \Rightarrow 2$, and the generated input 11. Also, it adds the link $w_{12} \rightarrow u_2$, and u_2 with its widgets into *f-UGF*, and returns *inloop* FALSE to indicate to *Mobolic* that there is no loop occurrence in the exercising procedure. As such, the current UI is now u_2 and the exercising procedure continues.

Step#4. *Mobolic* repeats Step#3 for u_2 , and u_3 . After exercising w_{31} on u_3 , the current UI is u_1 . Since u_1 is already in *f-UGF*, the link $w_{31} \rightarrow u_1$ will form a loop in *f-UGF*. Therefore, to eliminate the loop, *SAT* will not add the link $w_{31} \rightarrow u_1$ into *f-UGF*. Instead, *SAT* returns *inloop* TRUE to indicate to *Mobolic* that there is a loop occurrence in the exercising procedure. Afterwards, *Mobolic* performs the A* search in *f-UGF* to find the shortest path from the current UI u_1 to the lastly discovered UI u_3 . The found path includes $w_{12} \rightarrow u_2; w_{21} \rightarrow u_3$. Since u_3 is constrained by the user input from the editable widget e_{11} , *iScheduler* retrieves the previously generated user input from e_{11} , i.e., 11. *Mobolic* injects the input, replays the found path, and reaches the lastly discovered UI u_3 . Therefore, u_3 becomes the current UI, i.e., the UI which is currently displaying on an Android device screen.

Step#5. On the current u_3 , *Mobolic* finds the next unexercised non-editable widget, i.e., w_{32} . *SAT* generates the relevant UI event for w_{32} , e.g., “Click” event. However, exercising w_{32} leads to the same UI, i.e., u_3 , and thus, this transition forms a loop in the exercising procedure. Since u_3 is already in *f-UGF*, the link $w_{32} \rightarrow u_3$ will not be added into *f-UGF*. *SAT* returns *inloop* TRUE to indicate to *Mobolic* that there is a loop occurrence in the exercising procedure.

Step#6. After exercising w_{32} , *Mobolic* realizes that the UIs u_2 and u_3 are fully exercised. As such, *Mobolic* adds u_2 and u_3 with all their widgets into the set of the fully exercised UIs *feui*, and removes them from *f-UGF*. As a result, in *f-UGF*, *Mobolic* changes the lastly discovered UI to u_1 . Also, *Mobolic* realizes that the current UI displaying on the device is still u_3 , which already is not in *f-UGF*. Therefore, *Mobolic* generates the system event “Back”. Note that the “Back” event brings the app to any previously discovered UI in *f-UGF*. Without loss of generality, we assume that the event brings the app UI to u_1 . Thus, after the “Back” event, u_1 becomes the current UI.

Step#7. Even though all the non-editable widgets are fully exercised on all the discovered UIs u_1 , u_2 , and u_3 , *Mobolic* continues the exercising procedure since the decision nodes ① and ④ are not fully covered yet. From the stored symbolic constraints for e_{11} , *iScheduler* forms the symbolic path constraint pc , i.e., $\{(x > 5)\}$. However, *iScheduler* realizes that $pc \{(x > 5)\}$ has already been covered by the input 11 which has been generated in Step#3. As such, it negates the $pc \{(x > 5)\}$, and forms another one, i.e., $\{(x \leq 5)\}$.

Step#8. The *smartCE* generates a user input by solving the formed $pc \{(x \leq 5)\}$, e.g., it generates a user input equals to 4. *Mobolic* injects the input into e_{11} , and updates the corresponding symbolic constraints for e_{11} from the formed symbolic path constraint $pc \{(x \leq 5)\}$. So, the updated symbolic constraint for e_{11} is now $[(x \leq 5)]$.

Step#9. Next, *Mobolic* executes *SAT* to exercise w_{12} . Upon exercising w_{12} , *Mobolic* covers the execution path $1 \Rightarrow 4 \Rightarrow 6$, and executes *code#3* since the UID conditional statement $(x = 5)$ in the decision node ④ is not satisfied, i.e., $x = 4 \neq 5$. For the editable widget e_{11} , *SAT* stores the generated user input 4, and the newly discovered symbolic constraint $[(x = 5)]$ which is encountered on the execution path $1 \Rightarrow 4 \Rightarrow 6$. So, the editable widget e_{11} has now two new symbolic constraints $[(x \leq 5)]$ and $[(x = 5)]$. Note that the current UI u_1 has not been changed since only the user inputs, satisfying the condition $x > 5$, lead to discovering the new app UIs u_2 , and u_3 , while other user inputs do not change u_1 (see Figure 5.1, Section 5.3.2).

Step#10. Next, *iScheduler* forms the symbolic path constraint $pc \{(x \leq 5) \ \&\& \ (x = 5)\}$ from the stored symbolic constraints for e_{11} . The *smartCE* generates a user input by solving the formed $pc \{(x \leq 5) \ \&\& \ (x = 5)\}$. As a solution, it generates the user input equals to 5. Next, *Mobolic* injects the generated input into e_{11} , and executes *SAT* to exercise w_{12} . Since the UID conditional statement $(x = 5)$ in the decision node ④ is satisfied, i.e., $x = 5 = 5$, *Mobolic* covers the execution path $1 \Rightarrow 4 \Rightarrow 5$, and executes *code#2*. The current UI u_1 is not changed. Note that *Mobolic* does not update the corresponding symbolic constraints for e_{11} from the formed symbolic path constraint $pc \{(x \leq 5) \ \&\& \ (x = 5)\}$ since e_{11} already includes the same symbolic constraints from Step#9.

Step#11. Next, *Mobolic* realizes that the decision nodes ① and ④ are fully covered since all their corresponding UID conditional statements with all their sub-conditional statements are executed, and thus the current UI u_1 is fully exercised. So, *Mobolic* adds it with all its widgets into the set of the fully exercised UIs $feui$, and removes it from $f\text{-}UFG$. At this moment, *Mobolic* realizes that all the discovered UIs u_1 , u_2 , and u_3 are fully exercised, and thus *Mobolic* terminates.

5.6 Empirical Evaluation

Mobolic is designed to be a practical approach. Therefore it allows the automated testing procedure to notably shorten the overall exercising time, and increase the code coverage. To demonstrate the *Mobolic* performance in practice, we downloaded 10 real-world open-source Android apps from F-Droid,¹⁵ performed testing, and compared our results with several state-of-the-art automated GUI-testing approaches. The obtained results have provided a first evidence of *Mobolic* practicality (see Section 5.6.5). Among 10 apps, we have five with primitive, and five with non-trivial GUI structures whose code is UI-related, as well five apps which require specific user inputs, and five apps which may accept random user inputs. We selected apps with particular properties (complexity of GUI and textual user inputs) which could be sufficient to show adequacy and feasibility of our approach. In addition to the apps selection procedure, since our technique is primarily designed to demonstrate how exhaustively it can exercise the discovered app UIs via the code coverage measurement, we excluded the apps which happen to have an “unnecessary code” which surely cannot be executed by our technique. For example, (1) a system-related code (via broadcast receivers) which are triggered by receiving system events, however, such events are not supported by our technique; or (2) a platform version-dependent code, portions of which can only be executed on a specific version of the Android platform, or (3) a “dead code” which has been left in the app (could be an app bug or forgotten to be removed), and thus will never be executed. Considering such criteria for the apps exclusion from our experiment, we believe that we do not bias our results, and we do not favour our technique in any way. For further assurance of the validity of our results, we use the same set of the selected apps to compare the results of our technique with others such as Human (manual test), Monkey, ACTEve, MobiGUITAR, and Sapienz. Using the same set of apps, we demonstrate the capability of these tools showing via the code coverage how exhaustively they can explore an app GUI comparing to our technique.

To evaluate the *Mobolic* performance, we ran it on 10 downloaded apps, and compared the obtained code coverage results and exercising time with other prevalent approaches including manual testing (Human), and three automated ones Sapienz [14], ACTEve [8], and MobiGUITAR [2]. We chose Human for comparison since manual testing can provide the most intelligent inputs. We chose Sapienz as a representative of the multi-objective search-based testing, which shows the best performance comparing to the Monkey and Dynodroid tools involving a random GUI exploration strategy. Note that Sapienz extends the model-based testing by employing the search-based approach, and exploiting an automated random user inputs generation. We compared with ACTEve since it is only the tool which performs an automated concolic testing. Also, we compared

¹⁵<https://f-droid.org>

with MobiGUITAR as a representative which shows the best performance comparing to the A³E and *SwiftHand* tools involving a systematic GUI exploration strategy.

5.6.1 Experimental dataset

In this section, we provide a summary of the characteristics of 10 selected apps showing in Table 5.2. We provide an app “*category*”, “*name*” and “*version*” of the apps used in our experiment. In addition, we provide the app characteristics such as “LOC” (a total number of the *executable lines* of the Java code), “SGUI” (a total number of the *screens* in the app GUI), and “WGUI” (a total number of the *accessible widgets* for all the discovered screens on an app GUI).

The “SGUI” value consists of the summation of every rendered screen (i.e., an app UI hierarchy view) which are discovered in an app GUI as guided by the transitions existing between the screens. Note that “SGUI” may include duplicated screens since an app GUI may have transitions leading to the same screen multiple times. The “WGUI” value consists of the summation of every accessible UI widget found on every discovered screen in an app GUI. We define an accessible widget as the widget which can be exercised via an app GUI by triggering any of the events supported by *Mobolic* (see Table 5.1, Section 5.3.5). The value “LOC” is obtained by the EMMA tool (see Section 5.6.3), and the values “SGUI” and “WGUI” are obtained by *Mobolic* itself.

Table 5.2: Summary of characteristics of 10 apps tested

App category	App name	App version	LOC	SGUI	WGUI
Games	Guess*	0.12	46	3	4
	Hangman*	0.3.1-alpha	142	4	4
	Yahtzee*	1.1	496	14	51
	Cowsay#	1.4	243	55	183
	GM Dice#	0.1.6	560	160	259
	MunchLife#	1.4.3	161	5	16
Utility	Pedometer#	1.4.1	985	39	96
	Ringtone generator#	0.4	694	11	40
	Bodha Converter*	1.0	779	49	180
	Authenticator*	2.21	1937	19	50

* apps requiring specific (concrete) user text-inputs

apps accepting random user text-inputs

From Table 5.2, we can see that the selected apps fall into two categories “Games” and “Utility”. To give a sense of what each app is about, we describe each app as follows:

1. App “*Guess*”. Our aim as a user is to guess a hidden random number within [1, 100] range with eight trials.
2. App “*Hangman*”. The app randomly picks 1 of 40,000 English words from its

database. Our aim as a user is to guess the hidden word. The app allows the user to type only a single character at a time.

3. App “*Yahtzee*”. The app is a dice game. Generally, the game consists of some rounds and involves multiple players. To initiate the game, it should consist of at least one round and involve one player.
4. App “*Cowsay*”. It turns text into ASCII cows, with speech (or thought) balloons.
5. App “*GM Dice*”. It is a dice rolling application with a special focus on the 3D20 role-playing game (RPG) system.
6. App “*MunchLife*”. It is a simple counter application for keeping track of your level while you are playing the card game Munchkin.
7. App “*Pedometer*”. It is a step counter with speed, distance, steps-per-minute and text-to-speech.
8. App “*Ringtone generator*”. It generates Morse code ringtones in WAVE and iMelody formats.
9. App “*Bodha Converter*”. The app supports general (binary, octal, decimal and hex) conversions up to $(2^63 - 1)$ in their respective formats. It also supports ASCII characters conversion in the range of NUL to DEL, i.e., $(0 - 127)$.
10. App “*Authenticator*”. The app implements a one-time password (OTP) generators. When generating the OTP, it requires the user key which should not be shorter than a certain length.

5.6.2 Experimental environment

To run *Mobolic*, we use Android emulators on a Linux localhost. The Linux machine was running 64-bit Ubuntu 14 on a 4-core CPU-i5 with 16 GBs of RAM. For the development of *Mobolic*, we use the Java programming language (Java 7), and the *UI Automator* library from the Android SDK. To communicate with Android emulators, we use the Android Debug Bridge (*adb*) tool.

For our experiment and evaluation, we use Android emulators since it is a traditional and cost-effective solution. In addition, Android emulators can be freely accessed and are suitable for performing UI, stress and performance testing. However, *Mobolic* can also be deployed on physical Android devices since it does not require any instrumentation of the Android framework. By manual checking, we ensured that all our selected apps do not require any specific Android device hardware so that our test results are not affected by running them on Android emulator. As such, we configured each Android emulator as

shown in Table 5.3 and all other settings retained their default values. Also, we did not modify the Android platform, and instead, we used all its default system images.

Table 5.3: Android emulator device specifications

Parameter	Value
Device	Galaxy Nexus
Target	Google APIs (API Level 21)
CPU/ABI	Intel Atom (x86_64)
Hardware	Keyboard Present
RAM	2GB
VM heap	128MB
Front/Back cameras	None
Internal storage	500MB (with default content)
SD-card	200MB (with default content)

5.6.3 Experiment design

In this section, we describe a design of our experiment following the recommended guidelines [189, 190]. Here, we state *rationale* and *objectives* of this study, as well as we establish several *Research Questions* to answer.

Rationale: Why this research work has been done?

We have been investigating previous research works about automated testing of GUI of the mobile apps. From our investigations, we found that they were trying to address an issue with the poor code coverage while designing their approaches to be fully automated. However, eventually, the developed approaches have managed to achieve, on average, around 45–50%. As we found, the main obstacles to achieving high code coverage (in practice, achieving 85–90%, on average, is usually a good coverage) are (1) an *automated input generation*, and (2) *automated GUI exploration strategy* since manual human interaction is to be avoided. The previous works were trying to tackle these two main issues by applying different techniques to automate their approaches. For example, designing a novel automated input generation systems, or using dynamic symbolic execution for input generation, or improving random GUI-testing technique, or developing different applications of the model-based technique for systematic GUI exploration. However, upon publishing the works, the developed approaches did not demonstrate convincingly that the issues were indeed resolved, or at least addressed to a certain extent via a notable improvement.

We found that the previous research works do not address those two issues simultaneously. Instead, they either focus on improving an automated input generation, or a GUI exploration strategy. However, even proposing different solutions to address

either of two issues, the developed approaches were found to be not much practical. For example, in practice, many of the developed approaches usually require the astronomical amount of time for an app exercising, or automatically generated textual inputs are not sensible enough for a proper systematic GUI exploration. As such, to develop automated approaches, the previous research works rely on the user's availability to manually set, e.g., a termination condition for their testing tools since they are not able to stop the testing process automatically, or a certain limit for the depth exploration of an app GUI to mitigate the problem with exponentially growing exercising time, or suggest a user to manually provide sensible textual inputs during the testing procedure or before the procedure is started for the purpose of automatic replaying the user inputs.

From our investigations, we can see that the issues with the poor code coverage and inadequate automated input generation remain so that there are many other research works have been further conducted to propose alternative automated approaches. Therefore, we believe that there is still a great relevance and necessity in providing a solution which addresses both issues. As such, in our work, we propose an automated GUI-testing approach, namely *Mobolic*, which introduces (1) a novel **customated input generation** mechanism to eliminate human interaction in the testing process, and (2) uses our novel *f-UFG*¹⁶ model of an app GUI to efficiently and effectively perform a systematic GUI exploration in an automated manner. Therefore, these two novel aspects enable our approach to significantly increase the code coverage results, and shorten the exercising time which makes our approach more practical.

Objectives: What is expected to be achieved with this research work?

In this research work, we raise several objectives to achieve. We list them below as follows:

1. We are to develop an automated approach which allows, for mobile apps, to automatically generate functional UI tests with high coverage within a reasonable amount of time via a systematic GUI exploration. We expect that our proposed approach achieves high code coverage and significantly reduces the exercising time (in comparison with existing approaches).
2. We are to develop an automated approach which allows, for mobile apps, to automatically generate sensible user inputs which are relevant to an app UI. We expect that our proposed approach can find relevant inputs for many real-case scenarios so that human interaction is not required.
3. We are to perform a comparison of our approach with the state-of-the-art existing automated approaches, and manual one. We expect that our proposed approach,

¹⁶The *f-UFG* model is newly introduced in DroidDEV (see Chapter 4), and exploited by *Mobolic*.

on average, outperforms the existing automated, and manual approaches, while showing higher code coverage results and shorter exercising time.

Research questions: What knowledge will be sought or expected to be discovered?

In this work, we establish several *Research Questions (RQs)* to answer. We list them below as follows:

- RQ#1:** How does our *Mobolic* testing technique impact a code coverage in comparison with manual testing of the mobile apps GUIs ?
- RQ#2:** How does our *Mobolic* testing technique impact a code coverage in comparison with random testing of the mobile apps GUIs ?
- RQ#3:** How does our *Mobolic* testing technique impact a code coverage in comparison with dynamic symbolic execution of the mobile apps GUIs ?
- RQ#4:** How does our *Mobolic* testing technique impact a code coverage in comparison with model-based testing of the mobile apps GUIs ?
- RQ#5:** How does our *Mobolic* testing technique impact an exercising time in comparison with the state-of-the-art existing automated testing techniques of the mobile apps GUIs ?
- RQ#6:** How does our *Mobolic* testing technique impact an exercising time and code coverage in comparison with the naïve symbolic execution ?

To answer these *RQs*, we include Section 5.6.5 where we do analysis and interpretation of the experimental results.

5.6.4 Experiment operation

In this section, we explain how we do an experimental setup for Human, Sapienz, ACTEve, MobiGUITAR and *Mobolic*. We ran *Mobolic*, Sapienz, ACTEve and MobiGUITAR for all 10 apps on five Android emulators, and averaged the obtained code coverage¹⁷ results. For *Mobolic*, we did not set any time limit for the exercising procedure since it implements the self-terminative GUI exploration strategy. For Sapienz, ACTEve and MobiGUITAR, we configured them using the suggested configuration¹⁸ which is proposed by Choudhary *et al.* in the paper [15]. As suggested, we also set 60 minutes exercising time limit for each app for Sapienz, ACTEve, and MobiGUITAR.

¹⁷In *Mobolic*, we use the same approach for the Java code coverage collection as in DroidDEV (see Chapter 4, Section 4.6.2).

¹⁸<http://bear.cc.gatech.edu/~shauvik/androtest>

For Human, we selected five advanced Android users including two authors of our paper [99], and three students. We ensured that the selected students have knowledge in the testing of the mobile apps GUIs, and know their intricacies. We manually tested all 10 apps following the rules such as (1) run the testing apps on the same Android platform as *Mobolic*, (2) avoid installing any dependent apps if the testing app prompts, and (3) navigate back if the testing app goes to an external app. After testing is completed, we averaged the obtained code coverage results. To make a fair comparison of Human with *Mobolic*, we did not set any time limit for the manual exercising since it innately cannot be extremely long due to the human nature but reasonable for every testing app. So, according to our reasoning about the testing app GUI and its depth exploration, we exercised each testing app until it was visually possible to identify the newly discovered app UIs.

5.6.5 Experimental results

We evaluate the effectiveness of Human, Sapienz, ACTEve, MobiGUITAR, and *Mobolic* regarding achieved code coverage and exercising time. In Table 5.4, we show their obtained code coverage (columns 2–6), and exercising time (column 7) for *Mobolic*. Note that we do not report the exercising time for Human, Sapienz, ACTEve, and MobiGUITAR. In our experiment, we focus on a comparison of the effectiveness of *Mobolic* with Human, Sapienz, ACTEve, and MobiGUITAR regarding the achieved code coverage. As such, for these approaches, we only report the code coverage results. Since the exercising time is not reported for Human, Sapienz, ACTEve, and MobiGUITAR, we refer to Section 5.6.4 where we discuss how the exercising time was chosen. However, for *Mobolic*, we still report the exercising time for an evidence purpose to show that it does not take the astronomically long time to exercise the apps.

In Table 5.4, we show the code coverage which is computed as follows:

$$Coverage = \frac{Average_number_of_covered_basic_blocks}{Total_number_of_basic_blocks}$$

As for Human, the average number of covered basic blocks is computed by taking the average of the results given by five users. As for *Mobolic*, Sapienz, ACTEve, and MobiGUITAR, the average number of covered basic blocks is computed by taking the average of the coverage results given by five Android emulators.

To answer RQ#1: *Mobolic* vs Human (manual testing)

From Table 5.4, we can see that, on average, Human achieves lower code coverage with 78% as compared to *Mobolic* with 92%. In comparison with *Mobolic*, Human provides relatively low code coverage for only two apps “*Guess*” and “*Hangman*” due to their

Table 5.4: Code coverage for Mobolic, Human, Sapienz, ACTEve, MobigUTAR, and time taken by Mobolic

App name	Mobolic, %	Human, %	Sapienz, %	ACTEve, %	MobigUTAR, %	Mobolic, mins
Cowsay#	91	80	78	43	47	49.47
GM Dice#	87	78	56	41	51	58.44
MunchLife#	89	89	58	42	49	5.08
Pedometer#	86	81	77	36	58	19.13
Ringtone generator#	93	77	60	49	69	23.95
Guess*	96	36	32	24	26	3.47
Hangman*	100	58	35	21	19	8.29
Yahtzee*	99	99	44	19	35	6.48
Bodha Converter*	98	98	53	16	48	39.81
Authenticator*	83	88	41	28	33	5.95
Average	92	78	53	32	44	22

* apps requiring specific (concrete) user text-inputs

apps accepting random user text-inputs

specific app nature. In particular, the app “*Guess*” prompts a user to guess a hidden random number within [1, 100] range with eight trials. Similarly, the app “*Hangman*” prompts a user to guess a hidden word which is randomly chosen from the app database of 40,000 English words. Due to such the specific apps nature and human reasoning about them, it can be seen that Human will hardly guess the correct number or word within the given number of trials. Therefore, Human will likely be covering code for an execution path which always loses the game, and may never cover code for an execution path winning the game. In contrast, through the app instrumentation, *Mobolic* obtains the correct value which is chosen by the app for a user to guess. Therefore, knowing that correct value, *Mobolic* is capable of covering code for both execution paths which lead to winning and losing the game. For the remaining apps, *Mobolic* and Human perform similarly.

To answer RQ#2: Mobolic vs Sapienz (search-based testing)

From Table 5.4, we can see that, on average, Sapienz and *Mobolic* cover 53% and 92% of code, respectively. In comparison with *Mobolic*, Sapienz provides relatively low code coverage. For the “*Guess*” and “*Hangman*” apps, Sapienz achieves low coverage. Due to the underlying random-based input seeding mechanism, Sapienz is unable to exercise these two apps adequately since they require concrete user inputs. For the “*Yahtzee*”, “*Bodha Converter*”, and “*Authenticator*” apps, Sapienz gives higher coverage. However, these apps have complex GUI models and thus would require a systematic GUI exploration for achieving higher code coverage (in fact, it depends on the actual tool implementation). Therefore, for these apps, due to the underlying random-based GUI exploration strategy, Sapienz fails to provide high coverage as well.

For the other apps which are marked with the symbol (#) in Table 5.4, Sapienz achieves the highest coverage. The results are simply explained by the fact that these apps are good to exercise with any random inputs since they accept any user inputs. However, we realised that Sapienz obtains the lowest coverage for the “*MunchLife*”, and “*GM Dice*” apps. After a manual investigation, we found that the “*MunchLife*” app requires more complex UI actions such as “*Roll*”, and “*Press*” to trigger particular execution paths. Similarly, for the “*GM Dice*” app, Sapienz obtains low coverage due to a lack of support for the particular UI actions such as “*Scroll*”, and “*LongClick*”. In contrast, *Mobolic* supports a comprehensive list of the UI events, and a limited set of the system events (see Table 5.1, Section 5.3.5), as well as its improved systematic GUI exploration strategy with customised input generation, which could discover certain execution paths which are inaccessible to random-based GUI exploration techniques with a random input generation.

To answer RQ#3: Mobolic vs ACTEve (dynamic symbolic execution)

From Table 5.4, we can see that, on average, ACTEve and *Mobolic* cover 32% and 92% of code, respectively. In comparison with *Mobolic*, ACTEve provides significantly lower code coverage. We manually studied the nature of the underlying implementation of ACTEve to understand the reason why code coverage is low. We found that ACTEve explores apps starting from their entry points and employs symbolic execution for the entire app rather than for particular statements. In sharp contrast, *Mobolic* performs symbolic execution on only the UID statements rather than trying to symbolically execute all possible statements in the entire app. Therefore, in comparison with *Mobolic*, ACTEve may require a large amount of time to adequately exercise an app, and achieve high code coverage. As a result, in practice, achieving high code coverage for ACTEve for all but trivial apps, is generally impossible within a reasonable time limit, and thus ACTEve is a time-capped approach. Also, ACTEve does not scale beyond the event sequences consisting of more than four events, and thus ACTEve is also a depth-limited approach. In sharp contrast, *Mobolic* does not have such limitations, and forms event sequences of any length to reach a particular UID conditional statement, and does so within a reasonable time.

To answer RQ#4: Mobolic vs MobiGUITAR (model-based testing)

From Table 5.4, we can see that in comparison with *Mobolic*, MobiGUITAR provides significantly lower code coverage. On average, MobiGUITAR and *Mobolic* cover 44% and 92% of code, respectively. In particular, MobiGUITAR provides the lowest code coverage for the “*Guess*” and “*Hangman*” apps due to their specific nature, where the apps prompt a user to guess a hidden random number or word, respectively, within several trials. However, MobiGUITAR is not able to generate concrete user inputs, and thus it may never cover the execution paths which are constrained by these user inputs. For the rest of apps, MobiGUITAR shows better performance since the apps’ functionality does not mainly depend on the concrete user inputs, i.e., random user inputs can also be generated to discover unexplored app UIs.

MobiGUITAR has an option to provide concrete user inputs manually through the tool’s configuration file. However, in practice, it may require significant manual efforts since the provided user inputs will be relevant to only the specified app (due to the MobiGUITAR implementation). Therefore, the provided inputs cannot be reused by another app. That is, if the inputs are to be provided by a user, it will require the user to manually specify inputs for every app to be tested. In contrast, *Mobolic* is designed to automatically generate relevant user inputs through the customized input generation mechanism so that the constrained app functionality can be adequately exercised without human intervention.

To answer RQ#5: Mobolic vs state-of-the-art automated testing techniques (exercising time)

In our experiment, we run the automated tools Sapienz, ACTEve, and MobiGUITAR for 60 minutes each as discussed in Section 5.6.4. The authors of the paper [15] explain that for their experiment, they had to set a time limit for the exercising procedure as none of the tools was able to terminate automatically. Even though we believe that 60 minutes would be a sufficient time frame to demonstrate the performance (regarding the achieved code coverage) of the testing approaches which are used in our experiment. From Table 5.4, we can see that, on average, *Mobolic* achieves higher code coverage within a shorter time comparing to the other automated approaches. On average, for the Java code, *Mobolic* achieves 92% within the average time of 22 minutes, while Sapienz, ACTEve, and MobiGUITAR achieve 53%, 32%, and 44%, respectively, within the average time of 60 minutes.

Comparing the time taken by every app which is involved in our experiment, we can see that the maximum time is 58.44 minutes for the “*GM Dice*” app, and the minimum time is 3.47 minutes for the “*Guess*” app. So, the exact exercising time taken by *Mobolic* for each app is less than 60 minutes, while the coverage results are higher in comparison with the other tools. From such observations, we may conclude that, in general, setting a time limit as a termination condition for GUI testing tools is not a proper solution. We can see that even giving a large time frame (60 minutes for each app), it does not necessarily guarantee the high code coverage results. Based on the *Mobolic* performance, we believe that in order to achieve a better efficiency (i.e., a shorter exercising time), and efficacy (a higher code coverage), more sophisticated GUI-traversal algorithms, input generation mechanisms, and tools’ underlying implementation are needed.

To answer RQ#6: Mobolic vs naïve symbolic execution

In order to avoid a path explosion, *Mobolic* only performs symbolic execution on the UID statements. Having such the feature, we are interested to investigate how it may impact an execution time and code coverage while only performing symbolic execution on the UID statements. For this purpose, we compare *Mobolic* with ACTEve. ACTEve is a tool for automated concolic testing of mobiles apps, which uses naïve symbolic execution as an underlying technique. As such, to demonstrate an impact on code coverage and execution time by executing only the UID statements in *Mobolic*, we use the results of ACTEve as a ground-truth (i.e., baselines for code coverage and execution time), and compare them to the *Mobolic* results for code coverage and execution time. We compare the results of ACTEve and *Mobolic* in Table 5.5. For comparison, we use the same set of apps as listed in Table 5.4. For the ACTEve results, we measure code coverage every minute. If code coverage has not been increased at least on one covered “basic-block”

within a minute, we assume that the ACTEve code coverage results have converged to its maximum. Thus, we manually terminate ACTEve and report the time which was recorded one minute ago and the code coverage which was achieved by that time. For the *Mobolic* results, code coverage and execution time are taken from Table 5.4.

Table 5.5: Code coverage, and execution time for Mobolic and ACTEve

App name	Mobolic, %	ACTEve, %	Mobolic, mins	ACTEve, mins
Cowsay#	91	64	49.47	98
GM Dice#	87	55	58.44	81
MunchLife#	89	53	5.08	79
Pedometer#	86	49	19.13	85
Ringtone generator#	93	68	23.95	109
Guess*	96	28	3.47	88
Hangman*	100	31	8.29	111
Yahtzee*	99	44	6.48	83
Bodha Converter*	98	38	39.81	105
Authenticator*	83	39	5.95	69
Average	92	47	22	91

* apps requiring specific (concrete) user text-inputs

apps accepting random user text-inputs

From Table 5.5, we can see that, for ACTEve, the average coverage results have increased on 47% in comparison with its coverage given in Table 5.4 where the execution time was limited to 60 minutes. However, together with its increased coverage, the average execution time has also increased on 52%. The ACTEve results show that its implementation of naïve symbolic execution does not scale well on even the mid-sized apps. For such the apps, on average, ACTEve still gives relatively low coverage results while taking a considerable amount of time for execution. In comparison with the *Mobolic* results, we can see that ACTEve requires on 314% more time to achieve its maximum (converged) coverage. However, taking much longer exercising time, the ACTEve average coverage results are still lower on 96% than for *Mobolic*.

5.6.6 Comparison of Mobolic with other mobile GUI-testing tools

To further evaluate the performance of *Mobolic*, we also run it on the set of Android apps and perform a comparison of the code coverage results with the tools, which are used in Chapter 4. So, for the overall performance evaluation, we use all the Android apps and tools from this chapter and Chapter 4. We configure the respective tools from both chapters following the specifications which are described in Chapter 4, Section 4.6.1, and Section 5.6.4 of this chapter. We also set up an execution environment as described in Section 5.6.2 of this chapter. We summarise the results in Table 5.6, where we show the obtained code coverage (columns 2–8), and exercising time (columns 9, and 10) for

Mobolic and DroidDEV.

In Table 5.6, we split the results into two logical groups based on the apps used. For Group#1, we refer to the apps marked with (#), which can accept random user-inputs. The averaged values for Group#1 are indicated in a row with the title “**Average#**”. For Group#2, we refer to the apps marked with (*), which require specific (concrete) user-inputs. The averaged values for Group#2 are indicated in a row with the title “**Average***”.

For the apps in Group#1, we can see that, on average, *Mobolic*, DroidDEV, and Human (ManualTest) perform similarly, and achieve the highest code coverage results among the tools used. We note that, on average, *Mobolic* outperforms DroidDEV on 1.1% for code coverage while taking on 24% more time than DroidDEV. For the retaining tools, Monkey and Sapienz, MobiGUITAR and ACTEve perform similarly. With the same time given (60 minutes), on average, Sapienz outperforms Monkey on 1.4%, and ACTEve outperforms MobiGUITAR on 4% for code coverage, respectively.

For the apps in Group#2, we can see that, on average, *Mobolic* achieves the highest code coverage result among the tools used. Human (ManualTest) performs worse giving mid-range coverage, while DroidDEV is giving low coverage. On average, *Mobolic* outperforms DroidDEV on 102% for code coverage. However, it also takes on 117% more time than DroidDEV. For the retaining tools, Sapienz performs similar to DroidDEV giving low coverage, and MobiGUITAR, Monkey, and ACTEve give the very low coverage. With the same time given (60 minutes), on average, Sapienz outperforms MobiGUITAR on 28%, Monkey on 46%, and ACTEve on 86% for code coverage, respectively.

5.7 Discussion

In this section, we discuss the principal differences between *Mobolic* and symbolic execution. We describe the novel aspects brought by *Mobolic* into automated testing of the Android apps.

Mobolic uses the symbolic execution technique to generate concrete user inputs to force the exercising procedure to execute the app code for a certain program path. According to [191], nearly 99% of statements in the real-world programs are user-input-independent, and thus performing symbolic execution on these statements is not necessary. Therefore, *Mobolic* symbolically executes only the UID statements, while the other user-input-independent statements are executed concretely.

In sharp contrast, existing symbolic execution tools [8, 17, 18, 75] tend to execute all the program paths symbolically, i.e., 100% of all statements, what makes existing tools impractical due to the unacceptable time requiring to finish the testing procedure. Therefore, in practice, search of the program paths needs to be *depth*-bounded or *time*-

Table 5.6: Code coverage for Mobolic, DroidDEV, Human (ManualTest), Monkey, Sapienz, ACTEve, and MobiGUITAR and time taken for Mobolic, and DroidDEV

App name	Mobolic, %	DroidDEV, %	Human (ManualTest), %	MobiGUITAR, %	Monkey, %	Sapienz, %	ACTEve, %	Mobolic, mins	DroidDEV, mins
Doty#	100	100	100	56	100	100	95	2.01	1.21
AndroFish#	95	95	92	4	43	44	31	11.36	10.87
Core#	88	85	93	53	88	68	49	1.06	0.35
DivideAndConquer#	86	86	93	42	78	70	58	5.08	4.69
Coloring for Kids#	97	97	97	49	95	88	60	39.81	37.04
Acrylic Paint#	96	95	96	18	61	55	43	7.45	5.42
Clock Live Wallpaper#	94	94	95	2	5	91	68	1.07	0.62
Signal Generator#	98	99	97	79	96	93	51	10.38	8.95
Mytronome#	98	98	87	90	91	89	62	5.02	2.94
Instant Sound Effects#	99	99	97	78	96	78	46	9.03	7.73
Lock Pattern Generator#	88	88	91	41	74	66	38	8.11	5.03
Battery Circle#	87	87	89	51	58	50	34	3.08	1.69
Battery Dog#	89	88	87	12	35	48	27	14.87	12.24
World Clock#	98	98	97	88	95	89	71	39.43	16.12
Colour Clock#	97	97	95	92	98	95	88	4.01	3.45
MemoPad#	87	85	86	48	87	77	59	123.05	117.82
WikiAndPad#	86	87	86	24	73	70	48	10.08	8.67
Cowsay#	91	82	80	47	73	78	43	49.47	33.09
GM Dice#	87	82	78	51	56	56	41	58.44	44.52
MunchLife#	89	80	89	49	56	58	42	5.08	1.61
Pedometer#	86	80	81	58	72	77	36	19.13	18.32
Ringtone generator#	93	91	77	69	52	60	49	23.95	24.29
Average#	92	91	90	50	72	73	52	21	17
Guess*	96	49	36	26	48	32	24	3.47	1.28
Hangman*	100	79	58	19	26	35	21	8.29	3.88
Yahtzee*	99	3	99	35	3	44	19	6.48	0.15
Bodha Converter*	98	68	98	48	40	53	16	39.81	23.94
Authenticator*	83	36	88	33	21	41	28	5.95	2.43
Average*	95	47	76	32	28	41	22	13	6

* apps requiring specific (concrete) user text-inputs

apps accepting random user text-inputs

capped [20, 21]. It is worth noticing that since *Mobolic* uses the symbolic execution technique, it also copes with the symbolic execution problems such as *path explosion*, *path divergence*, and *complex constraints*. In particular, *Mobolic* mitigates the path explosion problem by executing symbolically only the UID statements. The UID statements comprise of a small portion of the entire app code, and thus it helps to avoid the path explosion problem in most cases. In contrast, existing tools execute all the program paths symbolically. Therefore, path explosion becomes a common problem, and to deal with it, they limit their search of the program paths either by depth or time.

In practice, symbolic execution also suffers from an untrustworthy implementation of existing Satisfiability Modulo Theories (SMT) solvers [192–194]. Therefore, path divergence and complex constraints become common problems. To mitigate these problems, we found a solution by using a reliable SMT solver such as the Symbolic Math Toolbox in MATLAB [188]. The Symbolic Math Toolbox provides functions for solving and manipulating symbolic math expressions. The symbolic engine in MATLAB is solid, and such proprietary solution sounds more trustworthy, reliable and could be less buggy than any other existing third-party open-source SMT solvers.¹⁹

Existing open-source SMT solvers only have a limited set of built-in theories because implementing a complete SMT solver may require overwhelming efforts. As a result, existing open-source SMT solvers do not support all mathematics and symbolic computations. Importantly, by using the symbolic engine in MATLAB, *Mobolic* is capable of solving symbolic constraints in the integer and decimal domains, while existing open-source SMT solvers solve symbolic constraints in the integer domain only. Also, *Mobolic* is capable of solving symbolic constraints in the textual domain, while existing open-source SMT solvers commonly do not solve string symbolic constraints.

In summary, *Mobolic* is capable of generating concrete inputs, in most cases, except for the highly complex symbolic constraints. Therefore, *Mobolic* addresses the problem stated in Section 5.3.1 by symbolically executing only the UID statements, and using a robust symbolic engine, the Symbolic Math Toolbox in MATLAB. In contrast, existing tools have limited capabilities of solving complex symbolic constraints due to an unreliable implementation of existing SMT solvers.

5.8 Threats to Validity

In our experiment, we identify three main *Threats (THs)* to the internal and external validity [190, 195, 196] of the obtained results for *code coverage* and *exercising time*. We describe threats in the list below, and propose solutions to mitigate or eliminate them (if possible):

¹⁹https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

TH#1: As a threat to the internal validity, it concerns about the *app code instrumentation* and *app code coverage collection*. Due to the human errors, in an app, there is a possibility that it has places with unreachable code, i.e., “dead code”, or somewhere it contains “unused code”, e.g., for future implementation, or forgotten to delete. Also, an app may implement the “version-dependent” code via conditional statements to support multiple versions of the Android platforms, or an app code can be “device-dependent”, i.e., a runtime behaviour on an Android emulator or actual device is different as the app may require specific physical device hardware components. So, if we instrument the Java bytecode without prior checks for such the code which is mentioned above, the EMMA tool will include such the irrelevant code in the overall code coverage. For example, instead of the overall number of executable lines of the Java code which is 1800 *LOC* (expected, i.e., true value), EMMA will report 2100 *LOC* (misleading, i.e., false value) so 300 *LOC* is the extra code which cannot be executed as explained above. Thus, such the threat may lead to an unexpected code coverage results, and potentially, to misleading conclusions about the performance of *Mobolic*. So, to exclude such the code from the app instrumentation and code coverage, in our experiment, we check the source code for the code as mentioned above with the aid of the Eclipse IDE for Android. As such, we ensure that the obtained code coverage results are calculated using true values of *LOC*.

TH#2: As a threat to the internal validity, it concerns about the *completeness of GUI exploration* and *obtained code coverage*. In our experiment, we expect that *Mobolic* discovers as many app UIs as possible in an existing app GUI model. However, such the threat is uncontrollable due to the nature of the performed testing, i.e., “black-box”, where *Mobolic* does not have prior knowledge about the number of possible app UIs existing in the GUI model, which theoretically could be discovered. This threat mainly exists due to the possible failures in identifying the similarity between two app UIs. If an indeed unique app UI is recognised as the previously discovered app UI existing in the *f-UFG* model, *Mobolic* will determine a loop in the model, and a new app UI will not be added into the *f-UFG* model, and thus it will not be exercised. Due to such the behaviour, *Mobolic* may overlook certain app UIs which indeed exist in the original app GUI model, but they were not discovered during the app execution. Also, the reliability of similarity identification between two app UIs depends on the quality of the underlying GUI ripper which is, by default, provided by the Android developers. Also, there is another case when not all possible app UIs can be discovered. It is due to the fact that an app may not have entered its certain states depending on an internal app logic implementation, or textual user inputs which are provided during the testing process. As such, certain app UIs

may not be discovered. As a result, our code coverage results may not reflect the absolute code coverage value which is theoretically possible to achieve in an ideal case.

TH#3: As a threat to the external validity, it concerns about the *execution environment*, *experimental dataset size*, and *exercising time*. The exercising time does not solely depend on the absolute execution time of automated approaches, but also it depends on an execution environment. To eliminate a factor of the execution environment, we run the testing tools on the same personal computer. However, such external factors as tools configuration (e.g., emulator parameters, testing tool parameters and others), internal implementation (e.g., it depends on the developer coding skills and code optimization), and testing techniques involved (e.g., random, dynamic symbolic, search-based, or model-based testing techniques) impact the exercising time. In addition, for certain apps, the exercising time may be dramatically increased depending on an app itself (e.g., the complexity of the app GUI, required user inputs or internal execution flow in the app). Even though the average exercising time for our experimental dataset may not be used “*as is*” to rely upon in general, but indeed, it gives reasonable sensing of how efficient the testing techniques could be.

5.9 Related Work

In this section, we provide an overview of the state-of-the-art automated GUI-testing tools which implement random, model-based, or dynamic symbolic execution testing approaches. To perform a multivariate analysis of the tools, we compare various aspects of them as shown in Tables 5.7, 5.8, and 5.9. Each table holds and compares related properties for the listed tools. For each table, *Column 1* indicates a name of the tool which implements an approach, and symbol ($\uparrow$) indicates an improvement of a particular property or feature of the *Mobolic* over the other listed tools.

Table 5.7 gives a comparison of the *specifications* of the testing tools. *Column 2* shows an underlying framework on top of which the tool is built. *Column 3* states ease of use of the tool, i.e., it states whether the tool works out of the box without any complex configurations or fixes, i.e., “Easy”, whether it required some efforts, i.e., “Medium”, or major efforts, i.e., “Hard”, to make it run. This judgment is solely based on our experience, and the evaluation does not have the value of a user study, mainly because most of these tools are just early prototypes. Note that at the moment of writing our paper [99], *Word2Vec* was not publicly available due to the developers’ company policy; therefore, our analysis of the tool is based on its paper content. So, for *Word2Vec*, we indicate ease of use as “Unknown” since we were not able to identify ease of use. *Columns*

Table 5.7: Comparison of specifications of mobile testing tools

Tool	Underlying framework(s)	Ease of use	API level	Device	Self-terminative ?	Termination condition
Monkey	UI/Application Exerciser Monkey	Easy	1+	Physical&Emulator	No	Number of injected events
A ³ E	Troyd&RERAN	Medium	1+	Physical&Emulator	No	Depth exploration of GUI-model
Dynodroid	MonkeyRunner	Easy	10	Emulator	No	Number of injected events
ACTEye	MonkeyRunner	Hard	10	Emulator	No	Execution time
MobiGUITAR	AndroidRipper	Hard	10+	Emulator	No	Execution time
SwiftHand	Chimpechat	Hard	16+	Physical&Emulator	No	Execution time
<i>Mobolic</i>	<i>UI Automator</i>	<i>Easy</i>	18+	<i>Physical&Emulator</i>	<i>Yes</i>	<i>Inmate exploration of GUI-model</i>
Sapienz	MonkeyRunner	Easy	19	Physical&Emulator	No	Execution time
Word2Vec	iOS Monkey (XCTest)	Unknown	N/A (iOS)	Emulator	No	Execution time

4, 5 indicate the minimum supported API level of the Android framework requiring the tool to run, and Android device on which the tool can be deployed, respectively. Note that *Word2Vec* is developed for the iOS platform. As such, for this tool, the API level attribute is not applicable, so we indicate “N/A”. *Columns 6, 7* state if the discussed tool is self-terminative, and what its termination condition is, respectively. If “No”, it means that the tool is implemented in such a way that it will infinitely continue an app execution unless a user manually specified a termination condition (i.e., the termination condition is user-dependent). For example, a user limits the exercising procedure by time (i.e., minutes, hours, days etc.), or by the number of events to be injected (i.e., UI and/or system events), or by depth of exploration in the app GUI-model, (i.e., a length limit of the event sequence to be injected) (*Column 7*). If “Yes”, it means that the tool can terminate automatically, and thus a user is not required to manually specify a termination condition (i.e., the termination condition is user-independent). For example, *Mobolic* innately terminates once it cannot discover any new app UIs in the given app GUI model, i.e., *Mobolic* has explored as many app UIs in the GUI-model as possible, and all the discovered UIs in *f-UFG* are fully exercised.

Self-termination and termination condition are coupled together in such a way that a termination condition determines if an automated tool can be self-terminative. After our experiments, we would conclude that *the termination condition* is one of the critical factors and important properties of automated tools to increase a chance of that the functionality of the app under test is covered as much as possible. Importance of the termination condition is simply explained by the fact that beforehand we are not aware how much, e.g., time, events, or GUI depth-exploration are needed to cover particular or all possible app functionality via an automated app GUI exercising. Note that here, we are discussing the “black-box” functional UI stress testing and not white-box unit testing where app internals are known to a user (tester) before it runs a test. In comparison with the other listed tools, *Mobolic* implements an improved termination condition which makes *Mobolic* self-terminative.

Table 5.8 gives a comparison of common *characteristics* of the testing tools. *Column 2* shows whether the tool requires the Android platform, and/or app under test to be instrumented prior to the app testing. Note this column indicates only instrumentation which is required as a part of the tool implementation to make it possible to test an app. It does not indicate instrumentation for any other purposes, e.g., for the code coverage collection, which is not required for the app testing. *Column 3* indicates what events are supported by the tool. Note that Monkey, Dynodroid and ACTEve support a limited number of system events since they are highly structured and vary depending on the Android platform and app itself. Therefore, there is no clear way how to universally generate them on all available the Android platforms and existing Android apps. *Column 4* shows in which manner the tool generates a user input. A *manual* user input could be

Table 5.8: Comparison of characteristics of testing tools

Tool	Instrumentation	Events	Input generation	Box	Approach
Monkey	N/A	UI&System	Automated	Black	Random-based
A ³ E	N/A	UI	Manual	Black&Gray	Model-based
Dynodroid	Platform	UI&System	Automated&Manual	Black	Random-based
ACTEve	Platform&App	UI&System	Automated	White	Dynamic symbolic
MobiGUITAR	N/A	UI	Manual	Gray	Model-based
SwiftHand	App	UI	Automated&Manual	Gray	Model-based
<i>Mobolic</i>	N/A	<i>UI&System</i>	<i>Customated</i>	<i>Black</i>	<i>Model-based</i>
Sapienz	App	UI	Automated	Black&Gray&White	Search-based
Word2Vec	N/A	UI	Automated	Black	Model-based

provided either during the app testing or could be a predefined set of desired user values which are stored in the tool’s configuration file for a particular app. An *automated* user input can be either randomly generated, or derived based on the heuristics which are collected from the app, e.g., the GUI-model, or the app code (e.g., source or bytecode), during or before the app testing. *Columns 5, 6* state in which box-mode the tool can perform the app testing, e.g., depending on the source code availability, and which GUI exploration strategy it implements as an underlying testing approach. Generally, black-box requires only executable source, while gray-box may require reversing of the executable source, e.g., for instrumentation. White-box requires source code to be available. Gray- and white-boxes may limit the tools practicability due to the source code unavailability or inability to properly reverse an executable source.

It is a fact that mobile apps GUIs are highly interactive, and commonly require sensible user inputs which are difficult to generate in an automated manner. Thus, we made a step ahead and introduced an improved automated input generation mechanism, namely *Customated*. In comparison with the other listed tools, customated input generation is a fully automated mechanism aiming to generate “*human-like*” user inputs. It consolidates concrete (via performing symbolic execution), user predefined (via analysing heuristics (textual attributes) of a relevant editable widget on the foreground screen), and random (numerical or textual) user inputs. As such, customated input generation enables *Mobolic* to more adequately exercise the app functionality in an automated manner.

Table 5.9 gives a comparison of *underlying GUI exploration strategies* of the testing tool. In Table 5.9, b is the branching factor, i.e., the number of subsequent paths which originate from a given app UI, and n is a depth, i.e., the number of app UIs in the path leading to an interested app UI. *Column 2* states which GUI explanation strategy is implemented in the tool. *Columns 3, 4* indicate which GUI search algorithm it involves, and what the worst-case search time complexity it has, respectively. For random-based GUI exploration strategies, GUI search and the worst-case values are not applicable, i.e., “N/A”.

Table 5.9: Comparison of GUI exploration strategies of testing tools

Tool	GUI exploration	GUI search	Worst-case
Monkey	Random	N/A	N/A
A ³ E	Systematic	Depth-first	$O(b^n)$
Dynodroid	Improved Random	N/A	N/A
ACTEve	Improved Systematic	N/A	N/A
MobiGUITAR	Systematic	Depth&Breadth-first	$O(b^n)$
SwiftHand	Systematic	Depth-first	$O(b^n)$
<i>Mobolic</i>	<i>Improved Systematic</i>	A^* ↑	$O(n)$ ↑
Sapienz	Improved Random	N/A	N/A
Word2Vec	Systematic	Depth-first	$O(b^n)$

In comparison with the listed tools, *Mobolic* significantly reduces search time in the *f-UFG* model by involving the informed search algorithm A^* [102, 183]. In *Mobolic*, we implemented the A^* search in such a way that its time complexity is $O(n)$, which is linear in the number of app UIs (n) on the path leading to the lastly discovered app UI. Thus, A^* enables *Mobolic* to build the shortest UI-path to quickly reach the lastly discovered app UI. As such, *Mobolic* can continue the exercising process from the lastly discovered app UI and exercise all its yet unexercised widgets, and to discover new, yet undiscovered, app UIs (if any). Such the improvement makes *Mobolic* efficient and effective regarding the exercising time and achieved code coverage.

Apart from our comparison, there is another research work [16] which performs statistical analysis, and comparison of various attributes and properties of multiple automated testing techniques for the Android mobile apps. The work offers a comprehensive, in-depth comparison list of the testing tools existing in the literature including those which are used in our chapter. It establishes a general framework which abstracts all common characteristics of the online testing techniques proposed in the literature. In particular, it shows how the framework can be used to design experiments aimed at performing objective comparisons among different online testing approaches, and how it helps to identify the influence of different tools' parameters on the performance of the testing techniques.

5.9.1 Random testing

Random testing [6, 63, 64] is the most simple approach to exercising mobile apps. Random testing typically starts by creating a simplified model (or without model) of the app under test. The built model can be used to generate random inputs and/or sequences of actions. The random testing can be *guided* or *unguided*. The unguided (undirected) random testing does not have heuristics to guide its search. The guided (directed) random testing extracts heuristics from the app under test to guide its search and possibly input generation, e.g., *feedback-directed* [65] or *adaptive* [66, 67] random testing. In practice, there are several testing tools such as Randoop [197], Artemis [198], Dynodroid [10], EvoDroid [103], and DART [75] which implement *feedback-directed* automated random testing with an event prioritizing mechanism for the mobile apps testing.

Monkey [53–55, 95] does not require any knowledge of an app GUI model. It merely generates a sequence of pseudo-random UI events such as clicks, touches, or gestures, as well as the limited number of system events. To run Monkey, a user requires to set the number of events to be injected. It terminates once all the events have been injected. Monkey does not consider any app state or UI transitions and performs random actions at random positions on an app UI. Therefore, the adequacy of Monkey could be affected by the density and/or the physical size of UI widgets on app UIs. Also, Monkey is not

able to generate relevant to the current app state input which is commonly required for the mobile apps due to their highly interactive nature of GUIs. Thus, generally, it is a challenge for random testing techniques to adequately exercise app functionality without an intelligent input generation.

Dynodroid [10] implements an *improved* random-based GUI exploration strategy compared to Monkey. It can either select the events which have been least frequently selected (*Frequency* strategy), or can take into account the context (*BiasedRandom* strategy), i.e., events which are relevant in more contexts will be selected more frequently. Compared to Monkey, Dynodroid generates the UI and limited set of system events which are relevant to the current app state. In addition, Dynodroid allows users to manually provide inputs (e.g., login/password) when concrete ones are required. Same as Monkey, Dynodroid does not terminate automatically. Therefore it requires a user to manually set a certain number of events to be injected as a termination condition.

5.9.2 Model-based testing

Model-based testing approaches [6, 7, 177, 199–202] use the UI model which is derived from an app GUI either manually or dynamically during the app exercising. Generally, model-based testing is the most suitable mechanism for guiding automated UI testing and is usually combined with random testing, or dynamic symbolic execution. For example, A²T² [138] implements a default random testing mechanism guided by an inferred app UI model. In particular, it relies on a GUI-crawler to mimic actual user events on an app GUI, and automatically infer a GUI model. The GUI-crawler builds the GUI model based on an extracted *Event-Flow Graph* abstraction of the fireable events on app UIs. That is, the crawling procedure operates using the fundamental entity “*Event*” to explore an app GUI. It uses the extracted events with their pre-conditions (event sequences), which are to be executed from root to leaves of the GUI tree, i.e., from first app UI showing upon the app launch to its subsequent app UIs. So, due to such the design of the GUI-crawler, it does not concern about two important innate issues existing in the automated GUI exploration. Firstly, it does not consider the problem of multiple app restarts [12]. Secondly, it does not consider re-exercising of the previously discovered app UIs multiple times (i.e., exercising the same app UIs more than only once depending on a GUI flow). In particular, in order to execute every extracted event sequence, the GUI-crawler requires multiple app restarts which could make the exercising process notably lengthy. In addition, the GUI-crawler does not consider re-exercising the same app UIs if they were previously discovered. So, the GUI-crawling algorithm may overlook certain app UIs, especially if they are located at deep levels in a GUI model.

MobiGUITAR [2] is an automated GUI-driven testing framework for the Android apps. *MobiGUITAR* is based on the three-stage principal such as “*observation*”, “*extraction*”,

and “*abstraction*” of the run-time state of UI widgets. The “*abstraction*” is a scalable state-machine model that together with test coverage criteria helps to automatically generate functional test cases. *MobiGUITAR* implements the *Breadth*-first and *Depth*-first search algorithms for traversing an app UI-model. It restarts the exercising procedure from the starting app state when it cannot find any new ones. In practice, an app restart can be time-consuming [12] for most real-world Android apps. *MobiGUITAR* requires major user efforts to manually configure multiple tool’s parameters [15].

A³E [11] aims to systematically discover new app states with high coverage via two techniques, the *Depth*-first and *Targeted* exploration to improve method and activity coverage, respectively. The *Depth*-first exploration is a dynamic approach which is based on the automated exploration of GUI activities and their widgets in a depth-first manner. It builds an abstract GUI model which represents each activity as a single app state, without considering different states of the UI widgets on the app activity. Thus, this abstraction may lead to missing certain app functionality which would be easy to exercise if a more concrete model was used. The *Targeted* exploration is a directed approach which allows it to explore activities more efficiently by generating relevant intents. It uses static bytecode analysis to build the Static Activity Transition Graph (SATG) of the app, and then it systematically explores the SATG graph while the app runs on a mobile device (real or emulator). By solely generating relevant intents, only the limited number of activities can be explored. Certain activities (e.g., a login activity) could be constrained by a specific user input so that all the subsequent activities will not be discovered unless a relevant user input is provided.

SwiftHand [12] aims to achieve code coverage quickly by learning and exploring an abstraction of an app GUI model. It learns the GUI model by exploiting execution traces generated during the testing process and applying the State-merging DFA (Deterministic Finite Automaton) induction machine learning algorithm. *SwiftHand* uses the learned GUI model to choose inputs that lead to yet undiscovered app states. Also, the built GUI model aims to minimise the number of app restarts by searching for paths to reach new app states using only UI inputs.

Sapienz [14] is an approach to automated Android testing which uses multi-objective search-based testing. It automatically explores and optimises event sequences, minimises their length, while simultaneously maximising code coverage and fault detection. Sapienz extends model-based testing by employing search-based exploration and exploits automated random user input seeding and multi-level app instrumentation. Sapienz minimises event sequence length and maximises other objectives which can be combined in a Pareto-optimal multi-objective search-based approach. By using the Pareto optimality, it does not sacrifice longer event sequences, when they are the only ones which find faults, nor where they are necessary to achieve higher code coverage. Through its use of the Pareto optimality, Sapienz progressively replaces such longer sequences with shorter ones when

their performance is equivalent.

Word2Vec [202] is a novel deep learning based approach which addresses a problem of automatic generation of the most *relevant* text inputs in a use case context, and which is applicable on a large scale as well. The *relevance* is specific to the natural-language semantics which only humans can understand. As such, this approach focuses on natural-language processing using a recurrent neural network (RNN) and *Word2Vec* models, where traditional automated input generation approaches such as symbolic execution are not applicable. For automated systematic GUI exploration, the developed approach uses XCTest, an iOS Monkey engine, which in turn, adopts the *Depth*-first search strategy. For automatic input generation, the developed approach leverages a combination of the RNN and *Word2Vec* models to find meaningful text inputs which are related to the current app context.

The approach demonstrates its effectiveness. It measures the effectiveness by computing screen coverage, i.e., it counts the number of different (unique) UI screens which have been explored within a fixed time limit. The results show that, regarding the achieved screen coverage, using the automated input generation, the RNN model outperforms the random one by 46%, whereas the combination of the RNN and *Word2Vec* models outperforms it by 60%. The difference between the RNN model and the combination of the RNN and *Word2Vec* models highlights the effectiveness of using *Word2Vec*.

5.9.3 Dynamic symbolic execution

Symbolic execution [161, 203, 204] is another software testing technique. Its objective is to systematically discover as many execution paths in a program as possible. The main idea of symbolic execution is to replace concrete values with symbolic expressions that can assume any possible value. Therefore, symbolic execution, theoretically, may explore all possible paths through the program and generate test cases to achieve high structural coverage. In practice, symbolic execution has severe problems such as *constraint complexity*, *path divergence*, and *path explosion* [19]. Furthermore, symbolic execution generally requires a user to instrument an app or execution environment before testing.

ACTEve [8] performs automated testing for Android apps using concolic execution. ACTEve explores the apps starting from their entry point, but does not aim for particular targets, and employs concolic execution at a level of the entire app rather than on individual event handlers. ACTEve symbolically tracks events from a point in the Android framework where they are generated up to a point where they are handled in an app. Also, ACTEve uses concolic execution for reasoning about low-level properties of events (e.g., coordinates of the UI widgets), which it can treat more abstractly by using the UI models.

5.10 Conclusion

In this work, we propose a novel automated GUI testing technique, namely *Mobolic*, to achieve high code coverage for the Android app testing by combining the online testing technique and customized input generation. We implement *Mobolic* for Android apps, considering their specific non-trivial structure and highly interactive nature of GUIs. We evaluate the performance of *Mobolic* on 10 real-world open-source Android apps, and compare it with prevalent approaches including manual testing (Human), and automated ones such as Sapienz, ACTEve and MobiGUITAR. Our experimental results show that, on average, *Mobolic* is capable of achieving high code coverage of 92% within the time limit of 22 minutes on average, while Human, Sapienz, ACTEve, and MobiGUITAR achieve 78%, 54%, 32%, and 44%, respectively, within the time limit of 60 minutes.

In our future work, we are planning to further improve the *smartCE* customized input generation. In particular, we will improve the accuracy and reliability of concrete input generation. Thus, we further improve its ability to solve more complex variations of symbolic string constraints, i.e., constraints whose solutions lie in the textual domain. In addition, we will expand a set of supported commonly used input types which are needed for user-predefined inputs. Also, we will expand the limited number of currently supported system events to enable *Mobolic* to exercise service apps as well, i.e., apps which do not have GUIs. As a result, we will improve our tool reliability and practicability, widen a range of mobile apps which could be exercised, and further increase code coverage approaching 100% of the covered Java app code.

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

In this thesis, we have addressed two major problems motivating our work:

1. Mobile apps GUIs have the non-trivial structure. Therefore, in practice, it is a challenging task to generate tests with high coverage in an automated manner via systematic GUI exploration.
2. Mobile apps GUIs are innately highly interactive. Therefore, in practice, it is a challenging task to generate non-trivial relevant textual user inputs for an adequate exploration of the mobile apps GUIs.

For addressing the problems mentioned above, in this thesis, we proposed an automated GUI testing technique, which aims to generate tests with high coverage via a systematic exploration of the apps GUIs and customized user inputs generation. Our testing technique uses the model checking technique and event-driven finite state machine abstraction for systematic exploration of the app GUI, and customized input generation which involves the UI heuristics to automatically generate relevant user-predefined or random inputs, and involves symbolic execution to generate concrete user-inputs.

Our testing technique performs an automated app exercising via the GUI exploration with active learning using our novel *f-UFG* model (i.e., an event-driven finite state model of the app GUI without loops). It automatically refines the *f-UFG* model during the exercising process and systematically traverses the model by implementing the informed search algorithm A^* .

Our testing technique takes into account the highly interactive nature of the apps GUIs. It implements our novel customized input generation mechanism which provides realistic user inputs which could be predefined by a user for certain input types, concrete user inputs, or random ones when the user-specific or concrete inputs cannot be generated.

We evaluated the performance of our testing technique on two sets consisting of 20, and ten real-world open-source Android apps. For the first set of 20 apps, our technique shows that, on average, it performs similarly to manual UI testing (Human) regarding the code coverage (91.5% vs 91.0%). Also, on average, our technique outperforms prevalent existing automated approaches such as MobiGUITAR (91.5% vs 44.9%, 15.76 mins vs 60 mins) and Monkey (91.5% vs 71.9%, 15.76 mins vs 60 mins) regarding the code coverage and exercising time, respectively. For the second set of 10 apps, our testing technique shows that, on average, it is capable of achieving the code coverage of 92% within the average time of 22 minutes, while Human, Sapienz, ACTEve, and MobiGUITAR achieve 78%, 54%, 32%, and 44%, respectively, within the fixed time of 60 minutes.

6.2 Recommendations for Further Research

For a further research, we would recommend several major directions:

1. Our testing technique supports only the UI (i.e., “user-like”) events generation. However, system-level events in the mobile environment also play an important role which allows the mobile apps to communicate with each other by sending and receiving user-independent data. A uniform generation of the system-level events is not a trivial task since they are highly structured and vary depending on the Android platform and app itself. As such, for a further research direction, we would suggest to investigate and propose an approach to uniformly generate the system-level events regardless of a specific Android platform or app.
2. Our testing technique builds a stateless (i.e., UI event-flow-based) app GUI model to avoid the state-space explosion. The stateless model does not capture internal app states but solely relies on the visual GUI changes in the app depending on the generated UI events. Thus, the stateless model may not cover certain app functionality simply leaving it untested. As such, for a further research direction, we would suggest to investigate and propose an approach on how to build a finite state-aware model of the app under test since mobile apps are innately state-sensitive.
3. Our testing technique implements the customized textual input generation. As a part of the textual input generation, our technique generates concrete (numeric or textual) user inputs by symbolically executing only direct user-input-dependent conditional statements. The direct user-input-dependent conditional statements compose a constraint where a user value is evaluated against a constant value, e.g., $x < z$, or $x > 5$, or $(x \neq 5) \&\& (y > 0)$, where x and y are user values, and z is a constant runtime value. In contrast, the *indirect* user-input-dependent conditional

statements compose a constraint where a user value is evaluated against another variable which is non-user-dependent, e.g., $x > y$, where x is a user value, and y is a random runtime value. Also, *indirect* user-input-dependent conditional statements can be used in a loop condition, or constrained by another condition, e.g., “*from $i = 1..x$ do if $i \% 2 = 0$ then $y++$ else $y--$ end; if $y > 0$ then ...*”, where x is a user value and decides y , or “*if $z > 0$ then if $x > 5$ then ...*”, where x is a user value and depends on z which is non-user-dependent. Our current solution does not guarantee execution of code portions which are constrained by the *indirect* user-input-dependent conditional statements. As such, for a further research direction, we would suggest to investigate and propose a more sophisticated approach on how to generate solutions which also can satisfy the *indirect* user-input-dependent conditional statements.

Author's Publications

1. **Y. L. Arnatovich**, L. Wang, N. M. Ngo, and C. Soh, "A comparison of Android reverse engineering tools via program behaviors validation based on intermediate languages transformation," *IEEE Access*, vol. 6, pp. 12382–12394, 2018.
2. **Y. L. Arnatovich**, L. Wang, N. M. Ngo, and C. Soh, "Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customized input generation," *Software: Practice and Experience*, vol. 48, no. 5, pp. 1107–1142, 2018. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2564>
3. **Y. L. Arnatovich**, N. M. Ngo, T. H. B. Kuan, and C. Soh, "Achieving high code coverage in Android UI testing via automated widget exercising," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2016, pp. 193–200.
4. **Y. L. Arnatovich**, T. H. B. Kuan, S. Ding, K. Liu, and L. K. Shar, "Empirical comparison of intermediate representations for Android applications," in *2014 26th Software Engineering and Knowledge Engineering Conference (SEKE)*, Jul 2014, pp. 205–210.
5. C. Soh, T. H. B. Kuan, **Y. L. Arnatovich**, A. Narayanan, L. Wang, "LibSift: Automated detection of third-party libraries in Android applications," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2016, pp. 41–48.
6. C. Soh, T. H. B. Kuan, **Y. L. Arnatovich**, L. Wang, "Detecting clones in Android applications through analyzing user interfaces," in *2015 IEEE 23rd International Conference on Program Comprehension (ICPC)*, May 2015, pp. 163–173.
7. K. Liu, T. H. B. Kuan, **Y. L. Arnatovich**, "Detecting anomaly in the usage of database attribute," in *2014 26th Software Engineering and Knowledge Engineering Conference (SEKE)*, Jul 2014, pp. 705–710.

Bibliography

- [1] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. *arXiv preprint arXiv:1402.4826*, 2014.
- [2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software*, 32(5):53–59, Sept 2015.
- [3] Cuixiong Hu and Iulian Neamtii. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST ’11, pages 77–83, New York, NY, USA, 2011. ACM.
- [4] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, Mar 2014.
- [5] Xun Yuan and Atif M. Memon. Using GUI Run-Time State As Feedback to Generate Test Cases. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE ’07, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.
- [6] Weiran Yang, Zhenyu Chen, Zebao Gao, Yunxiao Zou, and Xiaoran Xu. GUI testing assisted by human knowledge: Random vs. functional. *Journal of Systems and Software*, 89:76 – 86, 2014.
- [7] JJ Gutiérrez, MJ Escalona, and M Mejías. A Model-Driven approach for functional test case generation. *Journal of Systems and Software*, 109:214–228, 2015.
- [8] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE ’12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.
- [9] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, Stochastic Model-based GUI Testing

- of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 245–256, New York, NY, USA, 2017. ACM.
- [10] Aravind Machiry, Rohan Tahlilani, and Mayur Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.
 - [11] Tanzirul Azim and Iulian Neamtii. 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 and Applications*, OOPSLA’13, pages 641–660, New York, NY, USA, 2013. ACM.
 - [12] Wontae Choi, George Necula, and Koushik Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA’13, pages 623–640, New York, NY, USA, 2013. ACM.
 - [13] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.
 - [14] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.
 - [15] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, Nov 2015.
 - [16] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. A general framework for comparing automatic testing techniques of Android mobile apps. *Journal of Systems and Software*, 125:322 – 343, 2017.
 - [17] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, Dec 2008.

- [18] Kobi Inkumsah and Tao Xie. Evacon: A Framework for Integrating Evolutionary and Concolic Testing for Object-oriented Programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, pages 425–428, New York, NY, USA, 2007. ACM.
- [19] Saswat Anand. *Techniques to facilitate symbolic execution of real-world programs*. Georgia Institute of Technology, 2012.
- [20] Junaid Haroon Siddiqui and Sarfraz Khurshid. ParSym: Parallel symbolic execution. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 1, pages V1–405. IEEE, 2010.
- [21] Matt Staats and Corina Pasareanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 183–194. ACM, 2010.
- [22] Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE*, volume 3, page 260, 2003.
- [23] Bum chul Kwon, Waqas Javed, Niklas Elmqvist, and Ji Soo Yi. Direct Manipulation Through Surrogate Objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11*, pages 627–636, New York, NY, USA, 2011. ACM.
- [24] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct Manipulation Interfaces. *Hum.-Comput. Interact.*, 1(4):311–338, December 1985.
- [25] Jun Rekimoto. Pick-and-drop: A Direct Manipulation Technique for Multiple Computer Environments. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology, UIST '97*, pages 31–39, New York, NY, USA, 1997. ACM.
- [26] Ben Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology*, 1(3):237–256, 1982.
- [27] Ken Hinckley. *Haptic issues for virtual manipulation*. PhD thesis, University of Virginia, 1997.
- [28] Julie A Jacko. *Human computer interaction handbook: Fundamentals, evolving technologies, and emerging applications*. CRC press, 2012.
- [29] Andries van Dam. Post-WIMP User Interfaces. *Commun. ACM*, 40(2):63–67, February 1997.

- [30] Robert J.K. Jacob, Audrey Girouard, Leanne M. Hirshfield, Michael S. Horn, Orit Shaer, Erin Treacy Solovey, and Jamie Zigelbaum. Reality-based Interaction: A Framework for post-WIMP Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 201–210, New York, NY, USA, 2008. ACM.
- [31] Michel Beaudouin-Lafon. Instrumental Interaction: An Interaction Model for Designing post-WIMP User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 446–453, New York, NY, USA, 2000. ACM.
- [32] Don Gentner and Jakob Nielsen. The Anti-Mac Interface. *Commun. ACM*, 39(8):70–82, August 1996.
- [33] Cody W Hanson. Mobile devices in 2011. *Library Technology Reports*, 47(2):11–23, 2011.
- [34] Cody W Hanson. *Libraries and mobile services*. American Library Association, 2011.
- [35] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a Single Model and Test Prioritization Strategies for Event-Driven Software. *IEEE Transactions on Software Engineering*, 37(1):48–64, Jan 2011.
- [36] Neha Chaudhary and O Sangwan. Metrics for event driven software. *Int. J. Adv. Comput. Sci. Appl.(IJACSA)*, 7(1):85–89, 2016.
- [37] S. Herbold and P. Harms. AutoQUEST – Automated Quality Engineering of Event-Driven Software. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 134–139, March 2013.
- [38] Stephen Michael McKain, Ulziidelger Lobo, and Justin Wallace Saunders. Graphical user interface testing, November 2008. US Patent App. 11/805,295.
- [39] R. Mahajan and B. Shneiderman. Visual and textual consistency checking tools for graphical user interfaces. *IEEE Transactions on Software Engineering*, 23(11):722–735, Nov 1997.
- [40] Atif M Memon. GUI testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.
- [41] Brad A. Myers. User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.*, 2(1):64–103, March 1995.

- [42] Mary Jean Harrold. Testing: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 61–72, New York, NY, USA, 2000. ACM.
- [43] Steve McConnell. Daily build and smoke test. *IEEE software*, 13(4):144, 1996.
- [44] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, Feb 2001.
- [45] X. Yuan and A. M. Memon. Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, Jan 2010.
- [46] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312.
- [47] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.
- [48] Ibrahim K. El-Far and James A. Whittaker. *Model-Based Software Testing*. American Cancer Society, 2002.
- [49] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. Model-Based Testing of Reactive Systems, Advanced Lectures. *Lecture Notes in Computer Science*, 3472, 2005.
- [50] A. Pretschner. Model-based testing. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 722–723, May 2005.
- [51] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based Testing in Practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.
- [52] Mr J Praveen Kumar and Manas Kumar Yogi. A survey on models and test strategies for event-driven software. *Int. J. Comput. Eng. Res*, 2:1087–1091, 2012.
- [53] Noel Nyman. Using Monkey test tools. *Software Testing & Quality Engineering*, 29(2):18–21, 2000.
- [54] B. Hofer, B. Peischl, and F. Wotawa. Gui savvy end-to-end testing with smart monkeys. In *2009 ICSE Workshop on Automation of Software Test*, pages 130–137, May 2009.

- [55] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated Testing and Debugging of SAT and QBF Solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 44–57, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [56] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.
- [57] Daniel Paqué. From symbolic execution to concolic testing.
- [58] Koushik Sen. Concolic Testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, pages 571–572, New York, NY, USA, 2007. ACM.
- [59] Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *arXiv preprint arXiv:1610.00502*, 2016.
- [60] P. McMinn. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, March 2011.
- [61] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854, 2001.
- [62] Stefan Wappler and Frank Lammermann. Using Evolutionary Algorithms for the Unit Testing of Object-oriented Software. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, GECCO '05*, pages 1053–1060, New York, NY, USA, 2005. ACM.
- [63] Hamlet Richard. *Random Testing*. American Cancer Society, 2002.
- [64] Zhang Jian and Cheung S. C. Automated test case generation for the stress testing of multimedia systems. *Software: Practice and Experience*, 32(15):1411–1435.
- [65] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.
- [66] Liu Huai, Xie Xiaodong, Yang Jing, Lu Yansheng, and Chen Tsong Yueh. Adaptive random testing through test profiles. *Software: Practice and Experience*, 41(10):1131–1154.

- [67] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive Random Testing. In Michael J. Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, pages 320–329, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [68] Joonas Lindholm. Model Based Testing. *Helsinki, November*, 1:2006, 2006.
- [69] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.
- [70] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.
- [71] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.
- [72] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, Jun 1991.
- [73] Jan Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [74] Robert M Hierons, Jonathan P Bowen, and Mark Harman. *Formal Methods and Testing: An Outcome of the FORTEST Network. Revised Selected Papers*, volume 4949. Springer, 2008.
- [75] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05*, pages 213–223, New York, NY, USA, 2005. ACM.
- [76] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM.
- [77] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, et al. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.
- [78] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.

- [79] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.
- [80] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.
- [81] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339, August 2009.
- [82] David Adamo, Dmitry Nurmuradov, Shraddha Piparia, and Renée Bryce. Combinatorial-based event sequence testing of Android applications. *Information and Software Technology*, 2018.
- [83] W. Wang, S. Sampath, Y. Lei, and R. Kacker. An Interaction-Based Test Sequence Generation Approach for Testing Web Applications. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 209–218, Dec 2008.
- [84] Xun Yuan, Myra Cohen, and Atif M. Memon. Covering Array Sampling of Input Event Sequences for Automated Gui Testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, pages 405–408, New York, NY, USA, 2007. ACM.
- [85] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *2009 IEEE International Conference on Software Maintenance*, pages 211–220, Sept 2009.
- [86] G. A. Di Lucca and M. Di Penta. Considering browser interaction in Web application testing. In *Fifth IEEE International Workshop on Web Site Evolution, 2003. Theme: Architecture. Proceedings.*, pages 74–81, Sept 2003.
- [87] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing Combinatorics in GUI Testing of Android Applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 559–570, May 2016.
- [88] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining Model-based and Combinatorial Testing for Effective Test Case Generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*, pages 100–110, New York, NY, USA, 2012. ACM.

- [89] X. Yuan, M. B. Cohen, and A. M. Memon. GUI Interaction Testing: Incorporating Event Context. *IEEE Transactions on Software Engineering*, 37(4):559–574, July 2011.
- [90] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft ’15, pages 33–43, Piscataway, NJ, USA, 2015. IEEE Press.
- [91] Inês Coimbra Morgado and Ana C. R. Paiva. Mobile GUI testing. *Software Quality Journal*, Sep 2017.
- [92] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez. QBE: QLearning-Based Exploration of Android Applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 105–115, April 2018.
- [93] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’14, pages 204–217, New York, NY, USA, 2014. ACM.
- [94] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyanyk. CrashScope: A Practical Tool for Automated Testing of Android Applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, May 2017.
- [95] Google 2018, Android Monkey. Accessed: February, 2018.
- [96] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyanyk. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 33–44, April 2016.
- [97] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.
- [98] F-Droid 2018, F-Droid: free and open source catalogue of Android apps. Accessed: February, 2018.
- [99] Arnatovich Yauhen Leanidavich, Wang Lipo, Ngo Ngoc Minh, and Soh Charlie. Mobolic: An automated approach to exercising mobile application GUIs using

symbiosis of online testing technique and customized input generation. *Software: Practice and Experience*, 48(5):1107–1142, Feb 2018.

- [100] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-time Systems Using Uppaal. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing*, pages 79–94, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [101] Y. L. Arnatovich, M. N. Ngo, T. H. B. Kuan, and C. Soh. Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 193–200, Dec 2016.
- [102] Girish P Potdar and RC Thool. Comparison of Various Heuristic Search Techniques for Finding Shortest Path. *International Journal of Artificial Intelligence & Applications*, 5(4):63, 2014.
- [103] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.
- [104] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.
- [105] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM.
- [106] Muhammad Shafique and Yvan Labiche. A Systematic Review of Model Based Testing Tool Support. In *Technical Report SCE-10-04*, Carleton University, Canada, 2010.
- [107] Q. Xie and A. M. Memon. Studying the Characteristics of a ”Good” GUI Test Suite. In *2006 17th International Symposium on Software Reliability Engineering*, pages 159–168, Nov 2006.
- [108] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving Android Malware for

- Auditing Anti-Malware Tools. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 365–376, New York, NY, USA, 2016. ACM.
- [109] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü. AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 103–114, Sept 2017.
 - [110] Hammad Khalid, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 610–620, New York, NY, USA, 2014. ACM.
 - [111] K. Moran, M. L. Vásquez, and D. Poshyvanyk. Automated GUI Testing of Android Apps: From Research to Practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 505–506, May 2017.
 - [112] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.
 - [113] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing GUI Event Traces. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 422–434, New York, NY, USA, 2016. ACM.
 - [114] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 987–992, New York, NY, USA, 2016. ACM.
 - [115] Young-Min Baek and Doo-Hwan Bae. Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 238–249, New York, NY, USA, 2016. ACM.

- [116] Gavin A Rummery and Mahesan Niranjana. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, 1994.
- [117] Wei Song, Xiangxing Qian, and Jeff Huang. EHBDroid: Beyond GUI Testing for Android Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 27–37, Piscataway, NJ, USA, 2017. IEEE Press.
- [118] Markus Ermuth and Michael Pradel. Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 82–93, New York, NY, USA, 2016. ACM.
- [119] Wei Yang, Mukul R. Prasad, and Tao Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 250–265, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [120] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 864–880, New York, NY, USA, 2016. ACM.
- [121] Hailong Zhang and Atanas Rountev. Analysis and Testing of Notifications in Android Wear Applications. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE ’17, pages 347–357, Piscataway, NJ, USA, 2017. IEEE Press.
- [122] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99, May 2015.
- [123] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. PATDroid: Permission-aware GUI Testing of Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 220–232, New York, NY, USA, 2017. ACM.
- [124] C. Cao, C. Meng, H. Ge, P. Yu, and X. Ma. Xdroid: Testing Android Apps with Dependency Injection. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 214–223, July 2017.

- [125] J. Yan, T. Wu, J. Yan, and J. Zhang. Widget-Sensitive and Back-Stack-Aware GUI Exploration for Testing Android Apps. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 42–53, July 2017.
- [126] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static Window Transition Graphs for Android (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 658–668, Nov 2015.
- [127] Ziniu Hu, Yun Ma, and Yangyang Huang. DroidWalker: Generating Reproducible Test Cases via Automatic Exploration of Android Apps. *CoRR*, abs/1710.08562, 2017.
- [128] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real Challenges in Mobile App Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, Oct 2013.
- [129] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.
- [130] Alain Abran, JW Moore, P Bourque, R Dupuis, and LL Tripp. Software engineering body of knowledge. *IEEE Computer Society, Angela Burgess*, 2004.
- [131] James Bach. Exploratory testing explained. 2003.
- [132] J. Itkonen and K. Rautiainen. 2005 Exploratory testing: a multiple case study. In *International Symposium on Empirical Software Engineering*, pages 84–93, Nov 2005.
- [133] Cem Kaner, James Bach, and Bret Pettichord. *Lessons learned in software testing*. John Wiley & Sons, 2008.
- [134] J. Itkonen, M. V. Mantyla, and C. Lassenius. How do testers do it? An exploratory study on manual testing practices. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 494–497, Oct 2009.
- [135] J. Itkonen, M. V. Mäntylä, and C. Lassenius. The Role of the Tester’s Knowledge in Exploratory Software Testing. *IEEE Transactions on Software Engineering*, 39(5):707–724, May 2013.
- [136] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated Testing with Targeted Event Sequence Generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013*, pages 67–77, New York, NY, USA, 2013. ACM.

- [137] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato. A toolset for GUI testing of Android applications. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 650–653, Sept 2012.
- [138] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261, March 2011.
- [139] Google 2018, Android UI Automator. Accessed: February, 2018.
- [140] Sébastien Salva, Patrice Laurençot, and Stassia R. Zafimiharisoa. Model Inference of Mobile Applications with Dynamic State Abstraction. In Roger Lee, editor, *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015*, pages 177–193, Cham, 2016. Springer International Publishing.
- [141] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, July 2010.
- [142] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012.
- [143] X. Wu, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu. Testing Android Apps via Guided Gesture Event Generation. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 201–208, Dec 2016.
- [144] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Lu. User Guided Automation for Testing Mobile Apps. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 27–34, Dec 2014.
- [145] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. SIG-Droid: Automated system input generation for Android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 461–471, Nov 2015.
- [146] Sable Research Group. Soot: A framework for analyzing and transforming java and android applications, 2016.
- [147] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage Criteria for GUI Testing. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of*

Software Engineering, ESEC/FSE-9, pages 256–267, New York, NY, USA, 2001. ACM.

- [148] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based System Testing: High Coverage, No False Alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 67–77, New York, NY, USA, 2012. ACM.
- [149] D. Amalfitano, A. R. Fasolino, and P. Tramontana. Reverse Engineering Finite State Machines from Rich Internet Applications. In *2008 15th Working Conference on Reverse Engineering*, pages 69–73, Oct 2008.
- [150] James C. King. A New Approach to Program Testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233, New York, NY, USA, 1975. ACM.
- [151] K. Inkumsah and T. Xie. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 297–306, Sept 2008.
- [152] L. Baresi, P. L. Lanzi, and M. Miraz. TestFul: An Evolutionary Test Approach for Java. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 185–194, April 2010.
- [153] Paolo Tonella. Evolutionary Testing of Classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA’04, pages 119–128, New York, NY, USA, 2004. ACM.
- [154] A. Li, Z. Qin, M. Chen, and J. Liu. ADAutomation: An Activity Diagram Based Automated GUI Testing Framework for Smartphone Applications. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 68–77, June 2014.
- [155] Daniel R. Hackner and Atif M. Memon. Test Case Generator for GUITAR. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion ’08, pages 959–960, New York, NY, USA, 2008. ACM.
- [156] Florian Gross, Gordon Fraser, and Andreas Zeller. EXSYST: Search-based GUI Testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE ’12, pages 1423–1426, Piscataway, NJ, USA, 2012. IEEE Press.
- [157] Booch Grady, Rumbaugh James, and Jacobson Ivar. The unified modeling language user guide. *Reading: Addison-Wesley*, 1999.

- [158] M. E. Joorabchi and A. Mesbah. Reverse Engineering iOS Mobile Applications. In *2012 19th Working Conference on Reverse Engineering*, pages 177–186, Oct 2012.
- [159] T. Takala, M. Katara, and J. Harty. Experiences of System-Level Model-Based GUI Testing of an Android Application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 377–386, March 2011.
- [160] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon. Lightweight Static Analysis for GUI Testing. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 301–310, Nov 2012.
- [161] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov 2012.
- [162] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed Symbolic Execution. In Eran Yahav, editor, *Static Analysis*, pages 95–111, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [163] Patrice Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’07, pages 47–54, New York, NY, USA, 2007. ACM.
- [164] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY ’13, pages 209–220, New York, NY, USA, 2013. ACM.
- [165] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taint-Droid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.
- [166] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, MCS ’11, pages 21–26, New York, NY, USA, 2011. ACM.
- [167] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX security symposium*, pages 569–584, 2012.

- [168] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-Driven Compositional Symbolic Execution. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [169] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [170] Patrice Godefroid. Higher-order Test Generation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, pages 258–269, New York, NY, USA, 2011. ACM.
- [171] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’08, pages 206–215, New York, NY, USA, 2008. ACM.
- [172] Patrice Godefroid and Daniel Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA ’11, pages 23–33, New York, NY, USA, 2011. ACM.
- [173] Rupak Majumdar and Ru-Gang Xu. Reducing Test Inputs Using Information Partitions. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 555–569, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [174] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended Symbolic Execution on Binary Programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA ’09, pages 225–236, New York, NY, USA, 2009. ACM.
- [175] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys ’11, pages 301–314, New York, NY, USA, 2011. ACM.
- [176] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS ’11, pages 627–638, New York, NY, USA, 2011. ACM.

- [177] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*, pages 110–121, 2000.
- [178] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.
- [179] Mark H Linehan. Semantics in Model-Driven Business Design. In *Proc. 2nd International Semantic Web Policy Workshop*, pages 86–93, 2006.
- [180] Juhnyoung Lee. Model-driven Business Transformation and the Semantic Web. *Commun. ACM*, 48(12):75–77, Dec 2005.
- [181] M. A. Gamboa and E. Syriani. Automating activities in MDE tools. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 123–133, Feb 2016.
- [182] John Klein, Harry Levinson, and Jay Marchetti. Model-driven engineering: Automatic code generation and beyond. Technical report, Software Engineering Institute at Carnegie Mellon University, 2015.
- [183] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.
- [184] Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input Dependence Analysis via Graph Reachability. Technical report, Mountain View, CA, USA, 2008.
- [185] Karim O Elish, Danfeng Yao, and Barbara G Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Workshop on Mobile Security Technologies*, 2012.
- [186] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.
- [187] Shuai Hao, Ding Li, William G.J. Halfond, and Ramesh Govindan. SIF: A Selective Instrumentation Framework for Mobile Applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '13*, pages 167–180, New York, NY, USA, 2013. ACM.
- [188] MATLAB 2018, Symbolic Math Toolbox. Accessed: May, 2018.

- [189] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.
- [190] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [191] G. Li, K. Lu, Y. Zhang, X. Lu, and W. Zhang. Mixing Concrete and Symbolic Execution to Improve the Performance of Dynamic Test Generation. In *2009 3rd International Conference on New Technologies, Mobility and Security*, pages 1–5, Dec 2009.
- [192] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [193] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In *Brazilian Symposium on Formal Methods*, pages 23–36. Springer, 2009.
- [194] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.
- [195] Robert Feldt and Ana Magazinius. Validity Threats in Empirical Software Engineering Research – An Initial Survey. In *SEKE*, pages 374–379, 2010.
- [196] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. Validity Concerns in Software Engineering Research. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10*, pages 411–414, New York, NY, USA, 2010. ACM.
- [197] Carlos Pacheco and Michael D Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.
- [198] Shay Artzi, Julian Dolby, Simon Holm Jensen, Andreas Moller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 571–580. IEEE, 2011.
- [199] Peter Mehrlitz, Oksana Tkachuk, and Mateusz Ujma. JPF-AWT: model checking GUI applications. In *Proceedings of the 2011 26th IEEE/ACM International*

- Conference on Automated Software Engineering*, pages 584–587. IEEE Computer Society, 2011.
- [200] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, and Dianxiang Xu. A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258, 2013.
 - [201] Gwendolyn H Walton and Jesse H. Poore. Generating transition probabilities to support model-based software testing. *Software: practice and experience*, 30(10):1095–1106, 2000.
 - [202] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic Text Input Generation for Mobile Testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 643–653, May 2017.
 - [203] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.
 - [204] Sangharatna Godbole, Durga Prasad Mohapatra, Avijit Das, and Rajib Mall. An improved distributed concolic testing approach. *Software: Practice and Experience*, 47(2):311–342, 2017.
 - [205] Tarja Systä and Kaisa Väänänen-Vainio-Mattila. On empirical studies to analyze the usefulness and usability of reverse engineering tools. *STEP 2005*, page 107, 2005.
 - [206] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering, ICSE '00*, pages 47–60, New York, NY, USA, 2000. ACM.
 - [207] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP '12*, pages 27–38, New York, NY, USA, 2012. ACM.
 - [208] Julian Dolby. Program Analysis for Mobile: How and Why to Run WALA on Your Phone. In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle, MobileDeLi 2015*, pages 47–48, New York, NY, USA, 2015. ACM.

- [209] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.
- [210] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.
- [211] Thomas Kistler and Michael Franz. A Tree-Based Alternative to Java Byte-Codes. *International Journal of Parallel Programming*, 27(1):21–33.
- [212] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.
- [213] David Chisnall. The Challenge of Cross-language Interoperability. *Commun. ACM*, 56(12):50–56, December 2013.
- [214] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A. D. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72, Oct 2011.
- [215] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1844–1851, New York, NY, USA, 2013. ACM.
- [216] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou. Analysis of Android Applications' Permissions. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pages 45–46, June 2012.
- [217] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.
- [218] Jon Meyer, Troy Downing, and Andrew Shulmann. Java Virtual Machine. 1997.
- [219] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

- [220] James Gosling. Java Intermediate Bytecodes: ACM SIGPLAN Workshop on Intermediate Representations (IR'95). *SIGPLAN Not.*, 30(3):111–118, March 1995.
- [221] Raja Vallée-Rai. The Jimple framework. 1998.
- [222] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. *Compiler Construction: 9th International Conference, CC 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25 – April 2, 2000 Proceedings*, chapter Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?, pages 18–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
- [223] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99*, pages 13–. IBM Press, 1999.
- [224] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations, 1998.
- [225] Dan Bornstein. Dalvik VM internals. In *Google I/O developer conference*, volume 23, pages 17–30, 2008.
- [226] David Ehringer. The Dalvik virtual machine architecture. *Techn. report (March 2010)*, 4, 2010.
- [227] Jim Huang. Understanding the Dalvik virtual machine. *Google Technology User Groups, Taipei*, 2012.
- [228] Marko Gargenta. *Learning Android*. ” O'Reilly Media, Inc.”, 2011.
- [229] S Christey. The Infinite Monkey Protocol Suite (IMPS). 2000.
- [230] Kenneth Benoit. Linear regression models with logarithmic transformations. *London School of Economics, London*, 2011.
- [231] David Hull. Using Statistical Testing in the Evaluation of Retrieval Experiments. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '93*, pages 329–338, New York, NY, USA, 1993. ACM.
- [232] David J Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. crc Press, 2003.

- [233] Cole Davis. *SPSS for Applied Sciences: Basic Statistical Testing*. Csiro publishing, 2013.
- [234] Douglas C Montgomery. *Design and analysis of experiments*. John Wiley & Sons, 2017.
- [235] Robert L Mason, Richard F Gunst, and James L Hess. *Statistical design and analysis of experiments: with applications to engineering and science*, volume 474. John Wiley & Sons, 2003.
- [236] Cole Davis. *Statistical Testing in Practice with StatsDirect*, 2010.
- [237] Amit Mishra. Reverse Engineering: The Promising Technology. In *Covenant University Conference Proceedings, SEIS*, pages 1–25, 2010.
- [238] Wego Wang. *Reverse engineering: Technology of Reinvention*. CRC Press, 2010.
- [239] Eldad Eilam. *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.
- [240] C. Raibulet, F. Arcelli Fontana, and M. Zanoni. Model-Driven Reverse Engineering Approaches: A Systematic Literature Review. *IEEE Access*, 5:14516–14542, 2017.
- [241] H. Washizaki, Y. G. Guéhéneuc, and F. Khomh. A Taxonomy for Program Meta-models in Program Reverse Engineering. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 44–55, Oct 2016.
- [242] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.
- [243] David Cutting and Joost Noppen. An Extensible Benchmark and Tooling for Comparing Reverse Engineering Approaches. *International Journal on Advances in Software*, 8(1&2):115–124, 2015.
- [244] Lajos Jenő Fülöp. *Evaluating and Improving Reverse Engineering Tools*. PhD thesis, 2011.
- [245] L. J. Fülöp, P. Hegedus, R. Ferenc, and T. Gyimóthy. Towards a Benchmark for Evaluating Reverse Engineering Tools. In *2008 15th Working Conference on Reverse Engineering*, pages 335–336, Oct 2008.
- [246] S. Lamprier, N. Baskiotis, T. Ziadi, and L. M. Hillah. CARE: A Platform for Reliable Comparison and Analysis of Reverse-Engineering Techniques. In *2013 18th International Conference on Engineering of Complex Computer Systems*, pages 252–255, July 2013.

- [247] Y. G. Gueheneuc, K. Mens, and R. Wuyts. A comparative framework for design recovery tools. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–134, March 2006.
- [248] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato. A comparison of reverse engineering tools based on design pattern decomposition. In *2005 Australian Software Engineering Conference*, pages 262–269, March 2005.
- [249] Yauhen Arnatovich, Hee Beng Kuan Tan, Sun Ding, Kaiping Liu, and Lwin Khin Shar. Empirical Comparison of Intermediate Representations for Android Applications. In *SEKE*, pages 205–210, 2014.
- [250] B. Bellay and H. Gall. A comparison of four reverse engineering tools. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 2–11, Oct 1997.
- [251] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, pages 30–39, Oct 1998.
- [252] Jussi Koskinen and Tero Lehmonen. *Analysis of Ten Reverse Engineering Tools*, pages 389–394. Springer Netherlands, Dordrecht, 2010.

Appendix A

Comparison of Android Apps Disassemblers via Functional Testing of Mobile Apps GUI

Summary

In Android, performing a program analysis directly on an executable source is usually inconvenient. Therefore, a concept of reverse engineering has been adapted to enable a user to perform a program analysis on a textual form of an executable source which is represented by an intermediate language (IL). For Android, Smali, Jasmin, and Jimple ILs have been introduced to represent the executable *Dalvik* bytecode in a human-readable form. To use these ILs, we downloaded three of the most popular Android reversing tools including *Apktool*, *dex2jar*, and *Soot*, which perform a transformation of the executable source into Smali, Jasmin, and Jimple ILs, respectively. However, the main concern here is that an inaccurate transformation of the executable source may severely degrade a program analysis performance and obscure its results. To the best of our knowledge, it is still unknown which tool most accurately performs a transformation of the executable source so that re-assembled Android applications can be executed, and their original behaviours remain intact. Therefore, in this chapter, we conduct an experiment to identify one of three tools, which most accurately performs the transformation. We designed a statistical event-based comparative scheme and conducted a comprehensive empirical study on a set of 1,300 Android applications. We compare *Apktool*, *dex2jar*, and *Soot* via random-event-based and statistical tests to determine a tool which allows the re-assembled applications to be executed and evaluate how closely the re-assembled applications preserve their original behaviours. Our experimental results show that *Apktool* which uses Smali IL, performs the most accurate transformation of the executable source since the applications, which are re-assembled from Smali, exhibit their behaviours closest to

the original ones.

Introduction

Nowadays, Android is no doubt the most popular smartphone operating system. Android dominated the mobile market with 85.0% share in 2017Q1.¹ In fact, due to the highest market share, many Android applications (usually called “apps”) are being developed and published without performing an adequate program analysis. The adequate program analysis helps an app developer to automate a process of analysing behaviours of his program regarding its code correctness, robustness, and safety before it reaches end-users. Unfortunately, after publishing a program, its source code is usually not publicly available, and performing a program analysis directly on the executable source may be inconvenient, which may also be hindered by various code optimisation, and obfuscation techniques. Therefore, a transformation of the Android apps executables (i.e., *Dalvik* bytecode) into intermediate languages (ILs) is desired [205]. An IL is the lowest-level human-readable programming language which is automatically generated by reversing tools using a transformation of an executable source into its corresponding textual representation [206].

In this chapter, we use three of the most popular Android apps reversing tools including *Apktool*,² *dex2jar*,³ and *Soot*,⁴ which perform transformation of the executable source into Smali (*Dalvik Assembler*), Jasmin (*Java Assembler*), and Jimple (*Java Simplified*) ILs, respectively. Other than these, another tools have also been developed for reversing the Android apps executables. For example, *Androguard*,⁵ *Dexpler*⁶ [207] which is built on the top of *Dedexer*,⁷ *Androguard* which is built to support transformation from *Dalvik* bytecode into *Java*-like code for existing frameworks *WALA*⁸ [208] and *Soot* [209]. However, an inaccurate transformation of the executable source may decrease a program analysis performance, and obscure its results if the analysis was performed using an IL with inadequate quality and reliability [210–212]. In fact, a “good” IL must be capable of representing a source code without loss of information [213], and independent of any particular source or target language.⁹

The recent research works suggest that Smali [214–217], Jasmin,¹⁰ [218–220], and

¹<http://www.idc.com/prodserv/smartphone-os-market-share.jsp>

²<http://ibotpeaches.github.io/Apktool>

³<https://github.com/pxb1988/dex2jar>

⁴<https://github.com/Sable/soot>

⁵<https://code.google.com/p/androguard>

⁶<http://www.abartel.net/dexpler/>

⁷<http://dedexer.sourceforge.net>

⁸http://wala.sourceforge.net/wiki/index.php/Main_Page

⁹<http://cs.lmu.edu/~ray/notes/ir>

¹⁰<http://jasmin.sourceforge.net/about.html>

Jimple [207, 221–224] ILs represent a source code so that they all can be used to carry out various program analyses. For example, in static program analysis, these ILs can be used to perform the mutation and metamorphic testing, clones, “bad smells”, data and privacy leaks detection, “points-to” analysis, and others. In dynamic program analysis, these ILs can be used for code instrumentation to inject user-defined probes into the code.

In fact, a program analysis is fundamentally based on *Control*- and/or *Data*-flow graphs. However, extracting the flow graphs from an executable source is not a trivial task. For example, an executable source lacks many useful properties of high-level languages, and thus it introduces various challenges to the program analysis tools. Furthermore, various code optimisation and obfuscation techniques can further obscure the program flows. For example, an inaccurate transformation of an executable source may not adequately reflect the original program behaviours in respective program structures so that their corresponding *Control*- or *Data*-flow graph could be imprecise.

In this chapter, we are inspired to conduct a study to identify a tool which performs the most accurate transformation of the Android apps executables into its respective IL. We downloaded 1,300 Android apps from *Google Play* (by 50 top free apps from 26 different categories), and processed them as follows. With the aid of *Apktool*, *dex2jar*, and *Soot* tools, we disassemble the downloaded apps to obtain Smali, Jasmin, and Jimple ILs, respectively, and assemble the apps from the obtained ILs without making any modifications to their ILs content. We install the downloaded (original), and re-assembled apps on Android emulators, and perform functional GUI-testing using an event-based Monkey test. We analyse the obtained Monkey test results (the number of successfully injected events) by performing *one-sample* and *two-sample* statistical *t*-tests. As a result, we identify a tool which performs the most accurate transformation. Our experimental results show that the apps which are re-assembled from Smali behave closest to the original ones. Therefore, we suggest *Apktool* as the tool which performs the most accurate transformation of the Android apps executables.

In this chapter, we made the following contributions:

1. We develop a statistical event-based comparative scheme using the parametric statistical tests, and functional GUI testing to evaluate an accuracy of the Android ILs transformation via an automated program behaviours validation.
2. We apply our scheme on 1,300 closed-source Android apps, and compare the program behaviours of the Android apps which are assembled from Smali, Jasmin, and Jimple ILs with original ones.
3. We identify a reversing tool which most accurately performs a transformation of the Android apps executables into its respective IL.

The rest of this chapter is organized as follows. Section 6.2 introduces the *Dalvik*

virtual machine, *Dalvik* executable, and shows differences between Java virtual machine and Java executable (bytecode). Section 6.2 provides details of our experiment design. Section 6.2 presents our experimental results. Section 6.2 discusses related work. Section 6.2 concludes our chapter.

Dalvik Virtual Machine and Dalvik Executable

In this section, we describe the special features of *Dalvik* VM and stress its differences from standard *Java* VM. We also show that *Java* bytecode structure is different from *Dalvik* bytecode (executable), even most Android apps are being developed in *Java*.

Dalvik virtual machine

A *Dalvik* VM is not a *Java* VM. The *Dalvik* is a *register-based* VM in which Android apps are executed. Since both apps and system services on the Android OS are, generally, implemented in *Java*, the *Dalvik* VM has been developed, so that an Android device can run multiple VMs efficiently. Every Android app runs in its own process, with its own instance of the *Dalvik* VM and it is referred to as sandboxing apps. When an Android device is started, a single virtual machine process, called *Zygote* is created, which pre-loads and pre-initializes core library classes (*.libc*). Once the *Zygote* process has been initialised, it will reside and listen for socket requests coming from the runtime process, which indicates that it should generate new VM instances based on the existing *Zygote* VM instance. Thus, by spawning new VM processes from *Zygote*, the boot time of *Dalvik* VM is highly minimised. All other *Java* programs or services are originated from this process and run as their own process or threads in their own address space.

The *Dalvik* VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management. The core library classes (*.libc*) which are shared across the *Dalvik* VM instances are, commonly, only read by apps. When the (*.libc*) classes are needed, the memory from the shared *Zygote* VM process is simply copied to the forked child process of the app's *Dalvik* VM. Such behaviour allows it to maximise a total amount of shared memory while still restricting apps from overlapping with each other and providing security across apps and sandboxing individual processes. In Figure 6.1, we show an overall process of how the *Dalvik* VM is created and used in the Android system.

Traditionally, the *Java* VM (*stack-based* architecture) design is an instance of VM which has an entire copy of the core library class files and any associated heap, thus memory is not shared across multiple instances of the *Java* VM. Since the *Dalvik* VM is *register-based* architecture, it requires, on average, 47% less executed VM instructions than the *stack-based*. However, the *register-based* code is, on average, 25% larger

Figure 6.1: Dalvik virtual machine: Creation and usage

than the corresponding *stack-based* code, and this increases a cost of fetching VM instructions per unit of time. Notwithstanding, an overall performance of the *register-based* VM takes, on average, 32.3% less time than *stack-based* VM to execute standard benchmarks [225–227].

With the above-mentioned assumptions, *Dalvik* VM was developed to meet the following criteria [225, 226]:

1. Limited CPU speed.
2. Limited RAM amount.
3. No swap space.
4. Battery powered.
5. Diverse set of Android devices.
6. Sandboxed application runtime.

Dalvik executable

The *Dalvik* executable (*.dex*) file format is designed to meet the requirements of systems that are constrained in terms of memory (RAM) and processor (CPU) speed. Since

Android apps are, usually, written in *Java*, *Java* classes (*.java*) are compiled by a *Java* compiler (*javac* tool), which have been further transformed into a single file *classes.dex* by a standard *dex* compiler (*dx* tool which is, by default, included into Android SDK). After applying the *dx* tool, the *classes.dex* file has had only distant relationship with the *Java* bytecode (*.class*). In Figure 6.2, we show a difference in the process of translation original *Java* source code into the *Java* bytecode and *Dalvik* executable.

The *Dalvik* executable file design is primarily driven by sharing of data between running processes. The main difference between the *Java* bytecode (*.class*) and *Dalvik* bytecode (*.dex*) is that all classes of an Android app (*.class*) are packed into a single file *classes.dex*. Note that this is not simply packing, all the classes in the same *Dalvik* bytecode (*.dex*) file share the same *field*, *method*, *tables* and other resources. In particular, *Constant Pool* stores all literal constant values used within a class such as *string constants* used in code as well as *field*, *variable*, *class*, *interface*, and *method* names [225, 227]. In Figure 6.3, we show a difference in the internal structure of the *Java* bytecode (*.class* files) and *Dalvik* executable (*.dex*) file.

Figure 6.2: Difference in process of transformation Java source code (*.java*) into Java bytecode (*.class*), and Dalvik executable (*.dex*)

Experiment Design

In our experiment, we use four Android emulators with pre-installed Android 6.0 (Marshmallow, API Level 23), *Apktool*, *dex2jar*, and *Soot* Android apps reversing tools, Monkey tool which is an automated GUI exerciser,¹¹ and, by default, included in Android SDK,

¹¹<http://developer.android.com/tools/help/monkey.html>

Figure 6.3: Difference in internal structure of Java bytecode (.class), and Dalvik executable (.dex)

one-sample and *two-sample t-test* statistics from MINITAB.¹² The experiment was conducted on a 4-core CPU@3.10GHz with 32GB RAM machine running Windows 10 (64-bit).

Experimental environment

To communicate with Android emulators, we use Android Debug Bridge (*adb*).¹³ The *adb* is a versatile Command Line tool from Android SDK, which permits communication with a running Android instance [228]. The detailed Android emulator device specifications are shown in Table 6.1. To prepare our testing environment, we performed the following steps:

1. We installed the apps on the emulators with the aid of *adb*. Each emulator is for original apps and assembled from Smali, Jasmin, and Jimple ILs.
2. For the first time, we manually ran the apps to provide their required information (if any), e.g., *login/password*, *email*, *server port*, and other normal means of access. Such manual efforts are required to mitigate a threat to the validity of our results since, by nature, Monkey is not able to automatically generate such user inputs. However, we could not provide any information for *two-step authentication* (e.g., one-time password, verification code), since it is usually required during app runtime, and cannot be stored by an app for its future use.
3. On every emulator, we uploaded a gallery of the most popular audio/video files, pictures and office documents (see Table 6.1), in case if our testing apps require

¹²<https://www.minitab.com/en-us/products/minitab>

¹³<http://developer.android.com/tools/help/adb.html>

them. So, the testing process will not be interrupted due to an app exception “file not found”, or similar.

Table 6.1: Android emulator device specifications

Parameter	Value
Device	Galaxy Nexus
Target	Android 6.0 (API Level 23)
CPU/ABI	Intel Atom (x86_64)
Hardware Keyboard	Present
RAM	2GB
VM heap	128MB
Front/Back cameras	Emulated
Internal storage	500MB
SD card	200MB
Internal storage content:	
audio/video files	.mp3, .mp4, .avi
pictures	.jpeg
office documents	.doc, .xls, .ppt, .pdf

Tools

Apktool: Dalvik Disassembler

To conduct experiment on Smali, we chose a tool named *Apktool*. The next two commands are used to disassemble and assemble Android package (.apk), respectively:

1. *apktool d <apk_name>*
2. *apktool b <directory_with_decompiled_apk>*

When an Android app is assembled, the tool automatically creates *build/* and *dist/* sub-directories in the root *directory_with_decompiled_apk/*. The *dist/* sub-directory contains the fully assembled Android package.

Dex2jar: Java disassembler

The *dex2jar* is another tool which provides Jasmin. The *dex2jar* consists of four basic components:

1. *dex-reader* is designed to read the *Dalvik* executable (.dex) file.
2. *dex-translator* is designed to convert *Dalvik* executable instructions to a dex-ir format.

3. *dex-ir* is the format used by dex-translator, and designed to represent the *Dalvik* executable instructions.
4. *dex-tools* are developed to work with *Java* bytecode (*.class*) files.

The *dex2jar* tool uses its internal algorithms to manipulate a *classes.dex* file. To run disassembling/assembling process, the following code can be used:

1. *d2j-dex2jar -f -o <jar_name1> <apk_name>*
2. *d2j-jar2jasmin -f -o <directory1> <jar_name1>*
3. *d2j-jasmin2jar -f -o <jar_name2> <directory1>*
4. *d2j-jar2dex -f -o classes.dex <jar_name2>*

The first two instructions are used to reverse *classes.dex* file from an Android package into Jasmin. The last two instructions are used to re-assemble the obtained Jasmin code into *classes.dex*. For all these commands, the first parameter serves as an output, and the second one serves as an input for file/folder. As *dex2jar* assembles only the *classes.dex* file from Jasmin code, we need to directly replace it in the Android package. Note that the Android package has the same structure as a standard (*.zip*) archive, thus, we can replace *classes.dex* in the Android package without corrupting its internal structure. For that reason, we renamed the extension of Android package from *.apk* to *.zip*, and ran 7-Zip¹⁴ archiver as follows:

```
7z a <zip_name> classes.dex
```

Once the replacement process is finished, we renamed the file extension from *.zip* to *.apk*.

Soot: Java optimization framework

The *Soot* is a language manipulation and optimization framework for Java. *Soot* is capable of generating four ILs for Java code: *.baf*, *.jimple*, *.shimple* and *.grimp* [223, 224]. In this work, we focus on Jimple since only transformation from *Dalvik* bytecode into Jimple is currently supported. *Soot* automatically checks an appropriate API level specified in *AndroidManifest.xml* file, and uses a corresponding Android framework (i.e., *android.jar*) for reversing and assembling. We run *Soot* with specified arguments as follows:

```
-src-prec, <[jimple|apk]>
-process-dir, <apk_folder>
-android-jars, <android_api_jar_folder>
```

¹⁴<http://www.7-zip.org>

Since *Soot* assembles only a *classes.dex* file from Jimple, it requires replacing the *classes.dex* file directly in the Android package. Thus, similarly as for *dex2jar*, we replaced the assembled *classes.dex* file in each Android package without corrupting its internal structure.

Monkey: automated GUI exerciser

In our experiment, we use the Monkey tool from Android SDK to automatically generate and inject pseudo-random user and system events into the testing Android apps. Monkey does not know anything about the software which is being tested, and thus it simply performs random actions at random positions on an app GUI. For example, it randomly generates clicks, touches, or gestures, as well as certain system-level events which are formed using a uniform probability distribution or *Infinite Monkey Theorem* [229]. However, in practice, Monkey shows high-performance results regarding the achieved code coverage and failures detection rate. According to Choudhary *et al.* [15], Monkey appears the most effective and efficient automated GUI-testing tool comparing to the other relevant ones. In particular, the authors conducted a comprehensive study using 68 Android apps which were obtained from various app categories and compared Monkey with the other state-of-the-art tools.

To further justify a choice of Monkey test, we list the following critical quality risks which Monkey test addresses as suggested by Nyman [53]:

1. Input validation.
2. Functionality of GUI.
3. Transformation completeness and reliability.

In particular, after reversing apps, their original behaviours may be changed during a transformation process causing them to crash or hang. So, to expose any of the risks mentioned above, we perform a functional GUI-testing of re-assembled apps. However, it is practically impossible, within a reasonable time, to manually perform tests involving a human user on a large set of 1,300 apps. Thus, we decided to perform our testing by (1) choosing Monkey as the best option for an automated GUI-testing which is suggested by Choudhary *et al.* [15], and by (2) choosing Monkey as a sufficient test to reveal the quality risks as suggested by Nyman [53].

We use Monkey from the Android SDK. The default Monkey generally looks for the following conditions:

1. If Monkey is restricted to run in one or more specific packages, it disallows any attempts to navigate to any other packages and blocks them.

2. If a particular app crashes or receives any unhandled exception, Monkey will stop and report an exception.
3. If a particular app does not respond, Monkey will stop and report an exception.

To automate Monkey tests, we created a batch script file (*.bat*). For each app, our batch script includes the following command:

```
adb shell monkey -p <package> -v <events_count>
```

In our experiment, *<package>* parameter is used to identify a single app package accessible to Monkey without any dependencies from other packages. The *<events_count>* parameter value was manually set to be 10,000 pseudo-random events (i.e., maximum number of events to be injected).

Data transformation

In our experiment, we use parametric statistics. One challenge to use parametric analytic techniques is such requirement that a population from where samples are taken must be *normally* or *approximately normally distributed*. In fact, for many non-normal distributions, there is a possibility to transform their experimental data by applying a mathematical transformation. After the transformation, their data may be more similar to a normal distribution. For instance, a positively skewed distribution with a long positive tail (e.g., *log-normal*) may be transformed into a near normal distribution by taking the logarithm of the data values. The logarithmic transformation is also a convenient to transform a highly skewed variable into *normal* or *approximately normal distribution* [230].

Figure 6.4 shows a cumulative histogram of data distribution of the results obtained from Monkey tests for original apps, and the apps which are assembled from Smali, Jasmin, and Jimple. Analogously, Figure 6.5 shows the histogram of the logarithmic transformation of Monkey tests results. It is important to note that, in our experiment, *Monkey tests results* are referred to as a *number of successfully injected events* for each app tested.

From Figure 6.4, we can see that a distribution of Monkey tests results is *log-normal*. From Figure 6.5, we can see that the *log-transformed* data values of Monkey tests results are *approximately normally distributed*. Therefore, we can use parametric tests on the log-transformed data values of Monkey tests results. Importantly, we performed the logarithmic transformation of Monkey tests results in order to use parametric statistical tests since they have the following valuable advantages compared to non-parametric ones according to [231–233]:

1. Easier to use and interpret.

2. More efficient and accurate than their non-parametric counterpart.

Figure 6.4: Monkey tests results *before* log-transformation: Log-normal distribution of data values

Experimental scheme

For our experiment, we downloaded 1,300 Android apps from *Google Play*, and disassembled them using *Apktool*, *dex2jar*, and *Soot* to obtain Smali, Jasmin, and Jimple ILs, respectively. Afterwards, without making any modifications to the generated ILs' content, we assembled the Android apps using the generated ILs as a source. Next, we installed the re-assembled apps on the Android emulators, ran Monkey tests with long sequences of pseudo-random UI and system events, and performed parametric statistical tests, i.e., *one-sample* and *two-sample t*-tests. We use the Monkey tests results (the number of successfully injected events before the app hangs or crashes) as input for our statistical tests. We apply statistics to identify if there are any differences in program behaviours between the re-assembled and original apps. The *one-sample t*-test helps us to conclude whether a particular Android app has passed the Monkey test. The *two-sample t*-test helps us to identify whether the apps, which are assembled from Smali, Jasmin, and Jimple ILs have introduced any notable differences in their behaviours comparing to the original ones, e.g., crashes or hangs which are not observed in the original apps for the same Monkey tests.

Figure 6.5: Monkey tests results *after* log-transformation: Approximately normally distributed data values

In our experiment, every Monkey test contains five trials (replications) for every app category. Every trial contains the same set of 50 apps from a particular category, which are assembled from Smali, Jasmin, Jimple, and original ones. Note that Monkey generates a sequence of pseudo-random events based on a manually specified *seed* value, i.e., for the same *seed* value, Monkey generates the same sequence of pseudo-random events. So, to fairly compare the apps behaviours, within a trial, we run Monkey with the same *seed* value for the same set of 50 apps from a particular category. Therefore, performing five trials, we can statistically analyse, and compare the behaviours of the apps which are assembled from ILs, and the original ones regarding the number of successfully injected events. When the Monkey test is finished, we perform *one-sample t*-test to conclude whether the Monkey test has passed successfully for a particular app, and next perform *two-sample t*-test to identify how close this particular re-assembled app preserves its original behaviours.

For our experiment, we designed and applied the following statistical event-based comparative scheme:

1. Disassemble Android apps using *Apktool*, *dex2jar*, and *Soot* tools to obtain Smali, Jasmin, and Jimple ILs, respectively.
2. Assemble Android apps from the unmodified source of the obtained ILs using corresponding *Apktool*, *dex2jar*, and *Soot* tools.

3. Install Android apps on four Android emulators. Each emulator is for original apps and assembled from Smali, Jasmin, and Jimple ILs. Note that, to be installed on an Android device, Android apps must be signed with a developer's private key (it may be from a self-generated key pair). Optionally, Android packages can be aligned using a default Android *zipalign* tool for more efficient RAM access during the app execution.
4. Run Monkey tests for original apps, and those which are assembled from Smali, Jasmin, and Jimple ILs. After every Monkey test, we wipe the Android emulators reverting them into their initial state. For every trial, we set a unique *seed* value for Monkey.
5. Perform parametric *one-sample* and *two-sample* statistical *t*-tests.

Empirical Evaluation

In our experiment, we apply statistics since we cannot solely rely on raw results from the Monkey tests since an app may crash or hang for any other reason which is not originated from a Monkey test itself. Therefore, to identify a tool which performs the most accurate transformation of the Android executables, we carry out parametric *one-sample* and *two-sample t*-tests on the log-transformed data of the Monkey tests results.

One-sample parametric t-test

We perform *one-sample t*-test to identify whether a particular app has *passed* or *failed* the Monkey test. For that purpose, we choose the following criteria to verify [234–236]:

Hypothesis:

$$H_0: \mu = \mu_0$$

$$H_1: \mu < \mu_0$$

Test Statistic:

$$t_0 = \frac{\bar{y} - \mu_0}{S / \sqrt{n}}$$

where $\bar{y}$ is the sample mean; μ_0 is the true mean value; S is the sample standard deviation; n is the sample size. The statement $H_0: \mu = \mu_0$ is called the **null hypothesis** and $H_1: \mu < \mu_0$ is called the **alternative hypothesis**. The alternative hypothesis specified here is called a **one-sided alternative hypothesis** because it would be true only if $\mu < \mu_0$. To test a hypothesis, we compute the test statistic, and then reject or fail to reject the null hypothesis H_0 based on the computed value of the test statistic. Also, we specify the set

of values for the test statistic that leads to rejection of H_0 . This set of values is called the **critical region** or **rejection region** for the test. Two kinds of errors may be used when testing hypotheses. If the null hypothesis is rejected when it is true, a type I error has occurred. If the null hypothesis is not rejected when it is false, a type II error has been made. The probabilities of these two errors are given special symbols:

$$\alpha = P(\text{type I error}) = P(\text{reject } H_0 | H_0 \text{ is true})$$

$$\beta = P(\text{type II error}) = P(\text{fail to reject } H_0 | H_0 \text{ is false})$$

The general procedure in hypothesis testing is to specify a value of the probability of type I error α , often called the **significance level** of the test, and then design the test procedure so that the probability of type II error β has a suitably small value.

In our experiment, if H_0 is true, we conclude that the testing app has *passed* Monkey test successfully. If H_0 is not true, we accept alternative hypothesis H_1 , i.e., the testing app has *failed* Monkey test. Note that we calculated μ_0 based on the log-transformed data values obtained from Monkey tests. In practice, the value of the mean μ_0 specified in the null hypothesis is usually determined in one of three ways. It may result from past evidence, knowledge, or experimentation. It may also be the result of some theory or model describing the situation under study, or result of contractual specifications.

Parameters $\bar{y}$ and S are evaluated according to [234–236] as follows:

$$\bar{y} = \frac{\sum_{i=1}^n y_i}{n}$$

$$S = \left(\frac{\sum_{i=1}^n (y_i - \bar{y})^2}{n - 1} \right)^{1/2}$$

where y_i represents a sample, $i = \overline{1, n}$.

To interpret *one-sample t*-test results, we use the *P*-value approach with the following assumptions [234–236]:

1. If *P*-value is less than 5% level of significance (α), we would reject the null hypothesis H_0 in favour of alternative hypothesis H_1 .
2. If *P*-value is greater than 5% level of significance, we would not reject the null hypothesis H_0 because there is no evidence that the treatment median is less than μ_0 .

The *P*-value approach has been adopted widely in practice. It is used to state that the

null hypothesis was or was not rejected at a specified α -value (**level of significance**). This is often called **fixed significance level testing**. P -value is the probability that the test statistic will take on a value that is at least as extreme as the observed value of the statistic when the null hypothesis H_0 is true. Thus, P -value conveys much information about the weight of evidence against H_0 , and so we can draw a conclusion at any specified level of significance (α). More formally, we define P -value as the smallest level of significance that would lead to rejection of the null hypothesis H_0 . It is common to call the test statistic (and the data) significant when the null hypothesis H_0 is rejected. Therefore, we may think of P -value as the smallest level at which the data are significant. Once P -value is known, we can determine how significant the data are without the data analyst formally imposing a pre-selected level of significance.

In Table 6.2, for each category, we show the number of apps which failed the Monkey test. To explain the values in Table 6.2, let us assume that we have an original *Google Chrome* app from *Communication* category. After performing the Monkey test, and applying the log-normal transformation to the Monkey test results, we obtain data as shown in Table 6.3. Next, after performing *one-sample t-test* on the log-transformed data values of the Monkey test results, we obtain another results as shown in Table 6.4. Afterwards, using the P -value approach, from Table 6.4 we conclude that *Google Chrome* app has failed the Monkey test since the obtained P -value ($P = 0.007$) is less than the required level of significance ($\alpha = 0.05$). Therefore, we increment the value in the cell (in Table 6.2, column “*Original*” and row “*Communication*”) by adding “1” to the current number of apps which are failed the Monkey test. Analogously, we test every app from 26 categories for Smali, Jasmin, Jimple, and original ones, and calculate the number of apps failing the Monkey test.

After performing *one-sample t-test* for all Smali, Jasmin, Jimple, and original apps, in Table 6.2, we calculate a total number of failures for the apps which are assembled from Smali, Jasmin, and Jimple, and original ones. Using these values, we can compute the percentage (analytic values) to determine if there are any notable differences in program behaviours between the re-assembled apps and original ones regarding the number of successfully injected events. We compute the percentage for each tool as follows:

$$P_{Tool} = \left(1 - \frac{Tool_failures - Original_failures}{Total_apps} \right) \times 100\%$$

where $Tool_failures$ is a total number of failures for the apps which are assembled from Smali, Jasmin, and Jimple; $Original_failures$ is a total number of failures for the original apps, i.e., apps without any transformation of their executables; $Total_apps$ is a total number of apps tested. Note that since we use 1,300 Android apps in our study, the latter parameter is constant, i.e., $Total_apps = 1,300$. In particular, if we compute the percentage, for example, for Smali, we define the values of the parameters as follows:

Table 6.2: Number of apps failing Monkey test for 26 categories tested

No.	Category	Tool name-Intermediate language			Original
		<i>Apktool-Smali</i>	<i>Dex2jar-Jasmin</i>	<i>Soot-Jimple</i>	
1	Books Reference	4	15	16	3
2	Business	5	12	18	3
3	Comics	5	17	15	4
4	Communication	7	25	14	6
5	Education	7	15	27	5
6	Entertainment	5	30	21	4
7	Finance	3	34	36	2
8	Health Fitness	8	16	16	8
9	Libraries Demo	6	15	15	5
10	Lifestyle	3	27	17	2
11	Live Wallpaper	6	17	20	4
12	Media Video	7	9	11	7
13	Medical	8	12	10	5
14	Music Audio	4	22	10	3
15	News Magazines	10	18	15	7
16	Personalization	5	22	12	3
17	Photography	4	24	21	2
18	Productivity	7	25	14	4
19	Shopping	11	28	29	9
20	Social	6	26	30	3
21	Sports	3	16	10	3
22	Tools	5	22	26	4
23	Transportation	4	20	20	3
24	Travel Local	7	16	14	5
25	Weather	8	24	16	7
26	Widgets	9	19	17	8
Total number of failures		157	526	470	119
Original program behaviours preservation, %		~97	~69	~73	N/A

Table 6.3: Statistic of original *Google Chrome* app for five replications (Monkey tests)

Replication	Total injected events (maximum 10,000)	Log-transformed values of “Total injected events”
1	65	0.105762
2	345	0.106431
3	403	0.106494
4	308	0.106386
5	345	0.106431

Table 6.4: One-sample t-test results for original *Google Chrome* app

Replications	T-value	P-value	Significance level
5	-4.12	0.007	0.05

$Tool_failures = 157$, and $Original_failures = 119$ (see Table 6.2, intersection of row “Total number of failures”, and columns “*Apktool-Smali*”, and “Original”, respectively). Therefore, we compute the percentage for *Apktool* as follows: $P_{Apktool} = (1 - \frac{157-119}{1,300}) \times 100\% \approx 97\%$. Analogously, we compute the percentage for *dex2jar*, and *Soot*. For *dex2jar*, $P_{Dex2jar} = (1 - \frac{526-119}{1,300}) \times 100\% \approx 69\%$, and for *Soot*, $P_{Soot} = (1 - \frac{470-119}{1,300}) \times 100\% \approx 73\%$.

To verify the above-mentioned computations, in Figure 6.6, we graphically interpret the results from Table 6.2. There are four areas in Figure 6.6, which are referred to as SMALI, JASMIN, JIMPLE, and ORIGINAL. The vertical axis represents 100% STACKED FAILING AREA for every category. The horizontal axis represents APPLICATION CATEGORIES. To calculate the values for each failing area, for example, for *Book Reference*, we obtained the number of apps which have failed the Monkey test from Table 6.2. That is, four apps for SMALI, 15 for JASMIN, 16 for JIMPLE, and three for ORIGINAL. Summing all these values, it gives us 38 apps in total. Since Figure 6.6 represents 100% STACKED FAILING AREA, we assume that 38 apps are 100%, and distribute the failing apps accordingly in their respective areas. Analogously, for other categories, we obtain failing areas for SMALI, JASMIN, JIMPLE, and ORIGINAL. As a result, in Figure 6.6, we obtain the visual representation of differences in failing areas between SMALI, JASMIN, JIMPLE, and ORIGINAL. Comparing the failing areas, we can see that SMALI failing area is the closest to ORIGINAL, from which we hypothetically conclude that the apps, which are assembled from Smali, most accurately preserve their original behaviours in terms of the number of successfully injected events, comparing to those which are assembled from Jasmin, and Jimple. As a result, we identify a candidate tool which performs the most accurate transformation of the Android apps executables, i.e., *Apktool*. To experimentally verify our hypothetical conclusions, we perform *two-sample t-test*.

Figure 6.6: Number of apps failing Monkey tests for 26 categories tested

Two-sample parametric t-test

We perform *two-sample t-test* to identify whether the apps, which are assembled from Smali, Jasmin, and Jimple ILs have introduced any notable differences in their behaviours comparing to the original ones. In this section, we experimentally verify our hypothesis about the candidate tool which has been identified in Section 6.2, and hypothetical conclusions about Smali, Jasmin, and Jimple ILs are indeed valid.

For that purpose, we choose the following criteria to verify [234–236]:

Hypothesis:

$$H_0: \mu_1 = \mu_2$$

$$H_1: \mu_1 \neq \mu_2$$

Assumption:

$$\sigma_1^2 \neq \sigma_2^2$$

Test Statistic:

$$t_0 = \frac{\bar{y}_1 - \bar{y}_2}{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^{1/2}}$$

where $\bar{y}_1$ and $\bar{y}_2$ are the sample means of the first and second samples, respectively; S_1^2 and S_2^2 are the sample variances of the first and second samples, respectively; n_1 and n_2 are the sizes of the first and second samples, respectively; σ_1^2 and σ_2^2 are unknown and unequal variances of the first and second samples, respectively.

In our experiment, if H_0 is true, we conclude that a particular tool *performs* the most accurate transformation of the Android apps executables into its respective IL, and the apps which are assembled from such IL, *exhibit* their behaviours closest to the original ones. If H_0 is not true, we accept alternative hypothesis H_1 which implies that a particular tool *does not perform* the most accurate transformation of the Android apps executables into its respective IL, and the apps which are assembled from such IL, *do not exhibit* their behaviours closest to the original ones.

Parameters $\bar{y}_1$, $\bar{y}_2$ and S_1^2 , S_2^2 are evaluated according to [234–236] as follows:

$$\bar{y}_k = \frac{\sum_{i=1}^{n_k} y_{i_k}}{n_k}$$

$$S_k^2 = \frac{\sum_{i=1}^{n_k} (y_{i_k} - \bar{y}_k)^2}{n_k - 1}$$

where y_{i_k} represents a sample, $i_k = \overline{1, n_k}$, $k = \{1, 2\}$.

To interpret the obtained results, we use the P -value approach with the following assumptions [234–236]:

1. If P -value is less than 5% level of significance, the null hypothesis H_0 would be rejected in favour of alternative hypothesis H_1 .
2. If P -value is greater than 5% level of significance, the null hypothesis H_0 would not be rejected.

For *two-sample t-test*, we use the data values for *Apktool–Smali*, *Dex2jar–Jasmin*, *Soot–Jimple*, and Original which we obtained from *one-sample t-test* (see Table 6.2 in Section 6.2). However, we found that *one-sample t-test* results are *log-normally distributed*. Therefore, to use a parametric test, we need to have data values which are *normally distributed* or *approximately normally distributed*. So, similarly as described in Section 6.2, we first apply the logarithmic transformation method to obtain *approximately normally distributed* data values. Afterwards, using the log-transformed data values, we perform the parametric paired *two-sample t-test* on *Apktool–Smali*, *Dex2jar–Jasmin*, and *Soot–Jimple* with Original. In Table 6.5, we show the results of *two-sample t-tests* for *Apktool–Smali*, *Dex2jar–Jasmin*, and *Soot–Jimple*. Comparing the obtained P -values for each tool with the significance level, we can see that the obtained P -value ($P = 0.093$) for *Apktool* is greater than the required level of significance ($\alpha = 0.05$). Therefore, we conclude that *Apktool* performs the most accurate transformation of the apps executables since the apps, which are assembled from Smali, are indeed most closely preserve their original behaviours regarding the number of successfully injected events.

Table 6.5: Two-sample t-test results for Apktool, Dex2jar, and Soot

Tool	Total data values	T-value	P-value	Significance level
Apktool	26	−1.89	0.093	0.05
Dex2jar	26	−3.99	0.001	0.05
Soot	26	−3.45	0.002	0.05

Related Work

Multiple techniques can be used to facilitate researches and practitioners in a challenging task of program comprehension. One of these techniques is reverse engineering [237–239], a process of extracting and consolidating high-level design information from an executable source. Reverse engineering helps to analyse an executable source of a software system or program in order to identify its components and inter-relationships and generate corresponding reports or intermediate representations of the executable source in a textual or graphical form, usually at a higher level of abstraction [240–242].

Multiple studies have been conducted in the field, which perform a comparison and evaluation of various reverse engineering tools and approaches. Cutting and Noppen [243] introduce the benchmark, namely RED-BM (Reverse Engineering to Design BenchMark), which consists of a large set of Java sources for reversing, and a set of measures to evaluate and compare tools performance and effectiveness in a systematic manner. The benchmark facilitates the analysis of the reverse engineering tools based on their ability to reconstruct class diagrams. Also, it provides a set of measures which facilitate the comparison and ranking of the tools. To evaluate the performance of the benchmark, the authors used 12 commercial and open source reverse engineering tools. The performance results vary from 8.82% to 100% demonstrating the ability of the benchmark to differentiate between tools.

Fülöp *et al.* [244, 245] perform the evaluation of reverse engineering tools using the proposed benchmark, namely BEFRIEND (Benchmark For Reverse engInEering tools workiNg on source coDe), the benchmark which supports different types of the reversing tools, programming languages, software systems, and also enables the users to define their own evaluation criteria. The authors suggest that using their benchmark, the reverse engineering tools results can be evaluated and compared effectively and efficiently. For example, with BEFRIEND, the results of the tools from different domains recognising arbitrary characteristics of source code, and can be subjectively evaluated and compared with each other. The results could be from tools such as “design pattern” or “bad code smell” miners, “duplicated code” detectors, “coding standard violation” checkers, and others. Using this benchmark as a fundamental basis, the authors plan to create a universal, and generally applicable benchmark that can facilitate the evaluation, and comparison of various reverse engineering tools.

Lamprier *et al.* [246] propose CARE, a uniform platform for comprehensive evaluation of behaviour miners (reversing tools). The CARE is approach-independent, and only requires a set of artificial data (programs and execution traces) as input to produce a behaviour model such as Finite State Automata (FSA) which uses states to represent the execution flow of the program. These authors designed CARE platform with an ultimate goal to (1) provide a benchmarking mechanism for reverse engineering tools comparison, (2) allow comparison of the tools based on a type of programs and behavioural patterns, and (3) help users in choosing the most accurate reversing tool for their objectives.

Gueheneuc *et al.* [247] present a comprehensive comparative framework for design recovery tools. The framework is built upon the authors’ experience, and also extends existing frameworks. These authors evaluated the framework on two design recovery tools, namely Ptidej and LiCoR. The obtained results show that the comparative framework is well-defined, comprehensive, and universal. However, further validation is required to verify whether the framework enables an objective comparison of tools.

Arcelli *et al.* [248] perform a comparison of reverse engineering tools based on

the design pattern decomposition. The authors suggest that in reverse engineering, the recognition of design patterns provides additional information related to the rationale behind the implemented system design. In particular, the design patterns do not only provide information about how the architecture has been built, but also why it has been built in such a specific way. In their experiment, the authors formalise the design patterns according to the drawbacks they may represent. In turn, such formalisation leads to the identification of the sub-patterns which are recurring fundamental elements of the design patterns. So, using the sub-patterns, the authors analyse the role of sub-patterns by evaluating two reverse engineering tools FUJABA and SPQR. During the evaluation process, the main focus was on how sub-patterns are exploited by the system to define and detect the implemented design patterns. In order to emphasise the similarities and differences between the tools, in the case study, the authors used a Composite Design Pattern.

Arnatovich *et al.* [249] evaluate reversing capabilities of three reverse engineering tools based on the quality assessment of the untransformed apps' semantics preservation in the generated intermediate representations. For their evaluation, the authors use Smali, Jasmin, and Jimple intermediate representations. They obtained a set of 520 Android apps, re-assembled them with reversing tools using the intermediate representations, and applied randomised event-based testing with statistical analysis of the test results. Based on the obtained results, the authors suggest that Smali most accurately preserves the untransformed apps' semantics.

Bellay and Gall [250] performed an evaluation of four commercial reverse engineering tools. They used Refine/C, Imagix4D, SNIFF+, and Rigi tools which analyse C-code. The main focus was on the tools capabilities to generate graphical reports such as *Call*-graphs, *Control*- and *Data*-flow graphs. The authors investigated the capabilities of the tools by applying them to a real-world embedded software system. In their experiment, they used various assessment criteria for tools evaluation, e.g., the capabilities of parsing techniques, capabilities of generation of intermediate representations via analysis of properties of textual and graphical reports, capabilities of browsing and editing source code facilities, and general capabilities such as supported platforms, extensibility, and searching features.

Armstrong and Trudeau [251] evaluated five reverse engineering tools SNIFF+, Rigi, CIA, Dali, and Bookshelf. The main focus was on the tools' capabilities to extract an architectural design from the source code. In particular, they focused on the abstraction and visualisation of the software system components and their inter-relationships. Their study reports that Dali has the same visualisation capabilities as Rigi since it uses Rigi's graph editor, SNIFF+ has limited visualisation capabilities, and the Bookshelf and CIA tools have a performance issue while displaying large graphs so they should not be used for medium-to-large sized systems.

Koskinen and Lehmonen [252] identified a necessity to analyse systematically, and in

detail, the information retrieval capabilities provided by the reverse engineering tools. For that purpose, the authors compare ten the most popular reverse engineering tools Eclipse Java Development Tools (JDT), Wind River Workbench (WRW), Understand, Imagix4D, Creole, Javadoc, Javasc, Source Navigator, Doxygen, and HyperSoft. They evaluate the reverse engineering tools capabilities in terms of four aspects including the *data structures*, *visualization* and *information request specification* mechanisms, and *navigation features*. The authors report that, for *data structures*, 80% of the tools can generate *Call-graph*, *Class* diagram is provided by 70% of the tools, and only 30% of the tools can do an automated re-documentation. For *visualization* mechanisms, 90% of the tools can view the executable source in a readable form, while 80% of the tools mainly focus on *Call-graph* visualisations. For *information request specification* mechanisms, 80% of the tools implement a basic filtering, and expansion of the *Call-graph* contents. Majority of the tools provide traditional text search mechanisms, and regular expressions are supported only by 40% of the tools. Also, for *navigation features*, navigation capabilities among methods, any kinds of data elements, and browsing history are supported by 80–90% of the tools.

Conclusion

In this chapter, we developed a novel, statistical event-based comparative approach to examine the accuracy of the Android ILs transformation via an automated program behaviours validation. For our evaluation, we use Monkey to automatically generate sequences of GUI pseudo-random user and system events. We apply *one-sample* and *two-sample t*-tests to empirically identify a reversing tool which performs the most accurate transformation of the Android apps executables into its respective IL.

We evaluated our approach performance on 1,300 real-world Android apps which were obtained from 26 distinct categories on *Google Play*. We used the *Apktool*, *dex2jar*, and *Soot* reversing tools to perform a transformation of the Android apps executables into their respective ILs, i.e., Smali, Jasmin, and Jimple. We identify program behaviours differences in the apps, which are assembled from Smali, Jasmin, Jimple, and original ones, based on the number of successfully injected events. We found that *Apktool* performs the most accurate transformation of the Android apps executables since the apps, which are assembled from Smali, most closely preserve their original behaviours. Our results show that the apps which are re-assembled by *Apktool*, using Smali IL, preserve ~97%, while *dex2jar*, and *Soot* using Jasmin, and Jimple, preserve ~69%, and ~73% of their original behaviours, respectively.