Automated Debugging of Android Applications using Spectrum Based Fault Localization

A thesis submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Prajwal H G

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

August, 2022

CERTIFICATE

It is certified that the work contained in the thesis titled **Automated Debugging of Android Applications using Spectrum Based Fault Localization**, by **Prajwal H G**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Subhajit Roy

Prof Subhajit Roy
Department of Computer Science & Engineering
IIT Kanpur

August, 2022

Declaration

This is to certify that the thesis titled **Automated Debugging of Android Applications using Spectrum based Fault Localization**

has been authored by me. It presents the research conducted by me under the supervision of **Prof. Subhajit Roy**.

To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgements, in line with established norms and practices.

Prajwal

Signature

Name: **Prajwal H G**

Programme: PhD/**MTech**/MDes

Department: Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur 208016

ABSTRACT

Name of student: **Prajwal H G** Roll no: **17807481**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Automated Debugging of Android Applications using Spectrum Based Fault Localization**

Name of Thesis Supervisor: **Prof Subhajit Roy**

Month and year of thesis submission: **August, 2022**

Android applications have become part of an average person's daily life. Mobile devices provide the world at our fingertips, and we use them so much so that an average American at least opens his phone 11 times a day. Applications are what make mobile a helpful device. But if it crashes, it leaves a very bad impression on the user. So apps should be tested before publishing them. A lot of studies have been done on android testing, normally these tools which test android applications use coverage and the number of crashes found as the metric to judge the testing quality. This paper focuses on building a testing framework SpecDroid based on SBFL for debugging Android applications. It is a framework that uses rSandom testing along with SBFL. Rather than focusing on the number of crashes found this uses Wasted effort to judge the quality of testing. On testing with multiple applications from F-Droid it was found that we could generate good localisation of the seeded bugs, and dynamic coverage data from hooks work well enough to generate Spectrum in very less time.

This Project is dedicated to my family, who have constantly supported me throughout my life.

Acknowledgements

First, I would like to thank My Thesis Guide Prof. Subhajit Roy; even at times when I wasn't producing results, he never left hope in me and has guided my thesis till this point. Then I would like to thank Prantik Chatterjee a PhD student who has been a constant guide throughout my thesis. Then I would like to thank Ashankur Tripathi, Baldip Bijlani, Randeep Kumar Sahu, etc. Thanks for helping me in my Thesis. I would like to thank my friends Ashish Kumar, Rahul Kumar, Vaibhav Pratap, Chinmay etc who have been a constant support throughout my stay at IIT. Then I would like to thank all IIT Kanpur staff who have their invisible hands in making everything more manageable. Ultimately, I want to thank my *Mother*, without whom I don't know what I would be...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Android has a 71.45 percent market share and over 3 billion Mobile devices in usage worldwide[1]. Android is the most popular operating system mainly because it is open-source, unlike its other counterparts. The number of apps in the biggest app store *Google Play Store* is 2.87 million apps[2], and roughly 3 thousand apps are submitted to the Google play store daily. To further articulate how big the app market is, in 2023 global mobile app revenue is supposed to generate over 935 billion US Dollars [3]. In terms of daily usage, some study shows an American person spends an average of 2 hours and 54 minutes[4] on their phones each day. Over eleven times a day, at least 49 per cent of Android phone users open some app. Android apps are not only used for entertainment and socialising, but it is also used for more critical fields like banking, investment, and management, to mention some. The need for sophisticated apps has never been as much as it is now, and the number of Android applications has grown along with those having bugs. To show this fact, 37% of apps[5] in the Google store is of low quality(buggy). It means one in every app published on the Google app store is buggy. This is due to negligence by Android Developers in testing their application as it's time-consuming, expensive and needs a different set of skills compared to app development[6]. But a study shows 53% of people stop using an app if it crashed[7]. Hence testing should be considered an essential part of App Development.

Mobile app testing is a process which checks an application before it is released for functionality and usability. Testing helps us verify whether an app meets the required standard and if there are bugs. Testing a mobile app is highly complex due to the diverse nature of available apps and platforms. To a certain extent, manual testing is done for all apps, but this can rarely test for all possible cases; for example, to generate all possible scenarios for a simple app, if we have ten possible actions on a page(The screen displayed on an app) of an app, there may be ten such pages, and some pages may be interconnected or making a cycle, the number of possible action grows exponentially.

So automation of mobile testing is a necessity. Mobile automation testing, over the past decade, a lot of work has been done in this field, and many automated Android testing tools have been developed[8, 9, 10, 11]. They use different frameworks, which use different algorithms/techniques to test the apps. Some of these techniques are Random techniques[12], Model-based techniques[13], A/B testing techniques[14], and Reinforcement learning techniques[8]. Every tool which uses a certain technique has its own metric. Most tools use the number of crashes that occurred while testing and coverage as metrics to measure the tool's efficiency. Do these metrics give accurate information about the quality of an application? Yes, typically, if an app is tested for very high coverage and no bug is found, we can consider it a well-tested app. But in some cases, even after finding a crashed test case, it takes some effort to find and correct the bug. We need to put in extra work to find the actual reason for failure. When a crash occurs, we get the information(log) about the crash: the exception which caused it, the line number where the app crashed and the stack trace of the last action by a command line tool called **logcat**. But the line pointed in logcat may not be the line of the actual cause of the bug. This leads to us manually searching for the cause without direction. We build a new testing framework for tackling this issue.

Spec-droid is the new framework proposed in this paper. We try to experiment with the application of Spectrum Based Fault Localization (SBFL) in Android application testing. We perform all testing at method granularity (every user-defined method in the Android application source code is considered as a component in our test). After a run of Spec-droid on an AUT(Application Under test), we get a ranking of methods based on how likely they are to be buggy. This ranking, along with logcat data, helps us save time in debugging a buggy app.

1.1 Spectrum Based Fault Localisation

Spectrum Based Fault localisation (SBFL)[15] is a technique to find the locality of buggy components. There are many debugging techniques, but SBFL is good as it has a low execution time. SBFL is already widely used for debugging Java software. It is a widely studied topic, and many techniques [15] are used for debugging large code bases. [16].

A test suite X is a set of test cases x. Each test case corresponds to a run. The activity pattern of a x can be represented as an m-dimensional binary vector. Where the i-th element is 1 if it was executed in that test case, otherwise it is 0. A test suite X consists of n such test cases. It can be represented as a $n \times m$ binary matrix A, where each element a_{ij} (i-th row, j-th column) is 1 if the j-th component was executed in the i-th test case, otherwise it is set to 0. This matrix is called an Activity matrix A; where the columns represent the involvement pattern of the components, and rows correspond to the test cases. Every Activity matrix has an associated error vector E, which is also an n-dimensional binary vector. Each bit e_i of E, represents whether the i-th test case is a passing or failing test case. If $e_i = 0$, the i-th test case is passing and if $e_i = 1$ then the i-th test case is failing. The tuple (A, E) is called a **Spectrum *S***[15].

For every component, different SBFL techniques use the spectrum S to generate a score called *Suspiciousness.* The components are ranked based on their Suspiciousness score. Then the components will be examined by users in the decreasing order of their ranking until the actual buggy component is found.

Usage of SBFL techniques to find bugs in Android Testing is minimal to non-

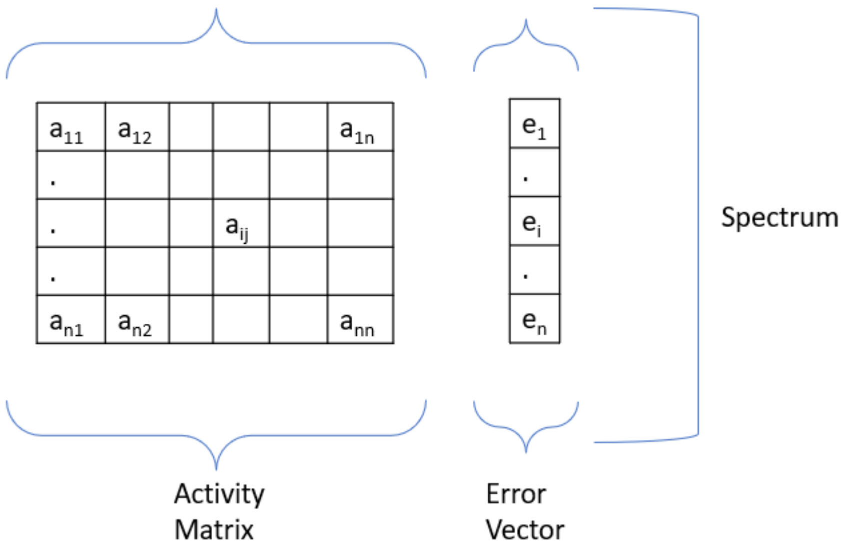


Figure 1.1: Spectrum

existent. The study of Spectrum S after executing multiple test cases has not been studied extensively for Android applications.

1.2 Android Activity and Widgets

A screen of an android app is called an activity, and an android app is made of many such activities. An activity is made of many widgets. There are many widgets such as buttons, text-box, etc. Users can perform different actions on these widgets based on their property and actions possible, for example:- clicking a button, inputting text to a text Field, averaging and dropping, etc. These actions are performed on a GUI, hence they are called *GUI Actions*.

GUI Actions:- An action(on GUI) can be shown as a tuple $A{=}(w,e,v)$, where w is a widget present in the activity; this is the action that can be performed on w, v is the value of text which can be used if any.
There are two types of widgets, namely actionable and non-actionable. Actionable widgets are widgets on which users can perform some actions. Some of them are buttons and text fields. Some actions that users can perform on them are click, long click, type, etc.

Non-actionable widgets are fixed widgets, and the user cannot perform any action on them in the present activity, for example, label, image view. Some actions which users can perform on these are select text, read text, etc.

GUI Activity State:- An activity can be denoted by a GUI state in the format of a k-tuple $GUI = (A_1, A_2, ..., A_k)$, Where A_i is the i^{th} GUI Action. k is the total number of actions available for the activity.
In an application, all the possible Activities and states can be achieved by taking GUI actions on a GUI State. GUI actions are the user's actions, and GUI state is the state of the AUT at any point in time.
An activity can be represented in XML format and parsed to a DOM (Document Object Model) for easier traversal. Every node in this DOM has the following attributes: package, class, text, resource-id, checkable, checked, clickable, enabled, focusable, focused, long-clickable, password, scrollable, bounds, and displayed.

1.3 Hooking an application

The term *hooking* implies a wide variety of techniques used to change or augment the behaviour of an application or software by intercepting events passed between components or messages or function calls. The code that handles such intercepted event is called a *hook*. In an android application, we inject code into the application. This is typically done when the application is running on an android device, but it can also be done before it is run. It is used to run custom code before, after or instead of the existing code.

1.4 Problem Statement

Given an Android application A and a failing test case B, we want to localise the component/components (method) that cause the crash.

1.5 Contributions

- We build a testing framework, SpecDroid for android applications.
- We develop an algorithm based on SBFL for debugging android applications using SpecDroid.
- We perform detailed experiments to evaluate the effectiveness of SpecDroid. We get a Wasted Effort of 0.007 and time 8 and half minutes per run for debugging our benchmark applications.
- We lay out future work that can be done using SpecDroid.

Chapter 2

Literature Survey

Every App testing framework roughly implements the following steps.

1. Install the Android Application Package(APK) in the device and undertake a static analysis of the Android application.

2. Generate test case either at once (pre-decide the input set of GUI actions) or generate a new action after an action is performed.

3. Observe the output and generate a new test case or report the output.

Over the years, many techniques[17] have been developed for testing, and some of them are Random Testing[12], A/B Testing[14], Concolic Testing[18], Mutation Testing[19], and Model-Based Testing[13]. In recent times Q-learning-based[8] approaches are being studied more. Also, these testing techniques may follow either black box testing, white box testing or grey box testing. Black box testing is used when we cannot access information about the code and the data structures used. White box testing is used when we have access to the source code, and grey box testing is used when the situation is in between these two cases. The testing techniques developed and studied for Android app testing are as follows:

Random Testing : Test is generated by selecting random widgets, and random independent inputs are used. Outputs are checked according to the specification given to differentiate between failure and success of the test case. Random Testing

is used by almost all techniques as a part of their testing methodology. For example, random testing is used at the start of the Q-learning-based approach.
Random testing is widely used due to its easy and simple-to-use nature and compatibility with different platforms, and it is also used in various industrial applications. An example for Random testing is: Android Monkey [12] is a tool in Android Software Development Kit that uses black box testing. A drawback of this testing method is the requirement of a very long sequence of events to increase coverage because many events may be redundant. For example, it may go between two activities repeatedly. Hence it requires a long time of execution to give meaningful results.

Mutation Testing : This technique is used to check the quality of the automated tests. It is a white box testing technique. The main algorithm goes as follows: a set of mutant operators are selected, and at certain points of source code, they are applied, one operator for each location. If we can find a mutated part of the program by a failing test case, then that mutant is said to be killed. If after all the automated tests are run and still the mutated app does not fail, then the set of automated tests is not up to mark.

A/B Testing : This technique [14] is also known as a controlled experiment. Here the new version of the app (treatment variant) is compared with the base version (control variant) to test which of the two variants is more effective. It is used generally for statistical hypothesis testing. These are normally done by testing a feature on a subset of users till we get statistically significant data. Many top companies like Google, Apple, and Facebook (Meta) use this type of testing in their apps.

Concolic Testing : This is a symbolic input test generation technique. It symbolically tracks events from the point where they originate to the point where they are ultimately handled. CONTEST[18] proposes a concolic testing technique to tackle the path explosion problem (the number of paths increases exponentially to check all possible paths).

Search-based Testing : It is a technique which uses a combination of meta-heuristic (problem independent) search techniques to provide an optimal test suite in terms of coverage or other diagnostic metrics[20]. Some search-based techniques are evolutionary testing[21], pareto multi-objective approach[22],etc . Evolutionary testing[21] is where an individual is a test case, and in a population with many individuals, the individual is evolved according to the corresponding metric (code coverage). Pareto multi-objective approach is used in Sapienz tool[22] which uses a combination of techniques. It has different models for the black, grey, and white box testing. The authors use black box testing for commercial apps that do not allow repackaging (also called skin testing), grey box testing when apk can be repackaged, and white box testing when source code is available. After the execution of a test, it uses a *StateMoniter* to monitor the execution states like coverage and covered activity and produces fitness scores, which are then used by a genetic algorithm to improve the score of the corresponding metric. They claim to optimize code coverage, sequence length, and the number of crashes.

Model Based testing : This generates test cases based on a model which describes the AUT. This method requires substantial manual effort to build the model. *Model-based Automated Testing of Mobile Applications* [13] uses model-based testing to automate test generation. The authors used a transition-based model with search-based test generation. The authors manually created a model on a high abstraction level. The model illustrates the application from user perspective; nodes are states and edges are actions which can be performed by the user.

Q-learning based Testing : It is a technique that uses Reinforcement Learning (RL)[8]. It uses a reward and punishment method to interact with the environment. Examples of RL Techniques: Actor-critic[23], Deep Q Network (DQN)[24],

App testing Techniques and Tools using them	
Name	**Tools/Techniques**
Random Testing	Dynodroid[26],Monkey[12]
Fuzzing Testing	IIntentDroid[10],IntentFuzzer[27]
Mutation Testing	EvoDroid[28]
Concyclic Testing	CONTEST[18]
Search Based Testing	Sapienz[22], EvoDroid[**evo**]
Model-Based testing	SmartDroid[29]
Q-learning based Testing	DroidbotX[8]

Table 2.1: Tools and App testing techniques used in them

State-Action-Reward-State-Action (SARSA)[25], and Q- Learning. DroitBox[8] uses a Q-learning-based test coverage approach to generate GUI test cases to maximize instruction coverage, method coverage, and activity coverage. The reward function is based on crashes and if the state is executed for the first time. Even though it is written, they focus on instruction coverage and method coverage. In a test case, the reward function is based on GUI activities rather than method coverage; hence, when the percentage of Activity coverage increases, the method and instruction coverage increases as well. This Q-learning approach using the upper confidence bound (optimism in face of uncertainty) approach and outperforms various state-of-art tools[8]. In [8], the authors study the usage of different state-of-the-art tools, and this study shows that Q-learning-based DroidbotX enhances the efficiency of tests compared to others (Droidbot, Humanoid, Sapiez, Stoat, Monkey). In contrast, Monkey, which uses Random Testing, performs many redundant events to get the same amount of coverage.

2.1 Frameworks used to build different tools

Some of the frameworks/programs used in making different frameworks/tools which does Android testing are: Monkey[12], appium[30], UI-Automator[31], A^3E[32], etc. **Monkey** is a test program released and maintained by Google. It generates a pseudo-random stream of events like click, fill text, gestures, etc. Monkey can be

directly used to stress test an app. Some tools use this feature as a base to develop either model-based, random, or search-based algorithms. Some of the tools which use it as a base are as follows AimDroid(Model-based/SARSA)[33], SmartMonkey(Random based)[12], Sapienz (Search based/Random)[22], AppDoctore (Random based), Dynodroid(Guided
/Random)[17], etc . This is one of the most used programs at present. **UI-Automator** is a framework which can be used to write code that interacts with any application installed on the device. UI-Automator gives APIs that can be used to develop UI Tests that can perform interactions on Apps for Android. [34] uses this as the basis to build a Q-learning-based algorithm. We use the Appium[30] framework for our work. Appium is an open-source test automation framework for native, hybrid, and mobile web apps. In [35], the authors design a Q-learning-based reinforcement learning testing algorithm which uses appium with UI-Automator to get the XML file of the present state of the app. Then the XML file is parsed to get widgets and the possible actions that the users can do on a widget. The authors use code coverage to measure the extent to which each test generation technique explores the functionality of the AUT (Application Under Test). A^3E[32] is also an android application GUI testing utility that facilitates developers to produce a similar sequence of interactions with the application an average user would do. It is aimed at modelling the application in the context of a Static Activity Transition Graph (STAG). A typical STAG shows the transitional relation between activities (Graph). A directional edge from activity A to activity B demonstrates a potential transition from A to B. It is used in a tool called Stoat[36], which was proposed by the authors in this paper.

Chapter 3

Tooling For Android Debugging

SpecDroid uses a combination of many tools/libraries for Android application testing. The primary tools used are Frida [37] for hooking the android app and the appium framework [30] for java to automate the test case generation process. Some other tools such as appium-inspector[38], dexdump[39], etc., are also used in building this framework SpecDroid . A brief description of the tools, along with some of their inbuilt functions that we use in this work, are described below.

3.1 Frida

Frida[37] is a dynamic code instrumentation toolkit for native apps. It allows a developer to inject snippets of JavaScript or other library snippets into native apps on Windows, macOS, GNU/Linux, iOS, Android, and QNX. Some simple tools built on top of Frida API are Frida-ps[40], Frida-trace[41]. Frida allows hooking in to live processes so that we can test, add functionality and debug such functions.
Some of the functions of Frida that are used in this framework are:

- Message handler (*on_message*): We can send messages from Frida-python to Frida-js using send and receive methods of this method handler.

- Get Methods (*get_Declared_Methods*): We can get declared methods of a hook using this function.

- hook (*Java.use*): This function is used to hook the class so that we can inject snippets of code into its methods.

Frida-python is the python code which manages the message handler to generate test case x of the test suite.
Frida-js is a javascript code which is run by Frida-python, this is used to hook the app.

3.2 Appium

Appium[30] is a test automation framework (open source) for use with native, hybrid and mobile web apps. The work done is on *Native Android Apps.* Several test automation tools like Robotium[42], Ranorex[43], etc., are also available with similar functionalities, but since Appium is the most popular test automation tool and has a large number of features available, it is used in this framework. Some of the user interactions with Android application widgets that we handle using appium in this project are:

- Click: This, as the name denotes, clicks a widget. It can also be used to toggle checkboxes.
- EditText: Using Appium, we input words into a text-box similar to keyboard interaction.
- Scroll: This interaction scrolls the page from one co-ordinate to other.
- back: This interaction presses the back button in the Android device/emulator.

3.3 Appium Inspector

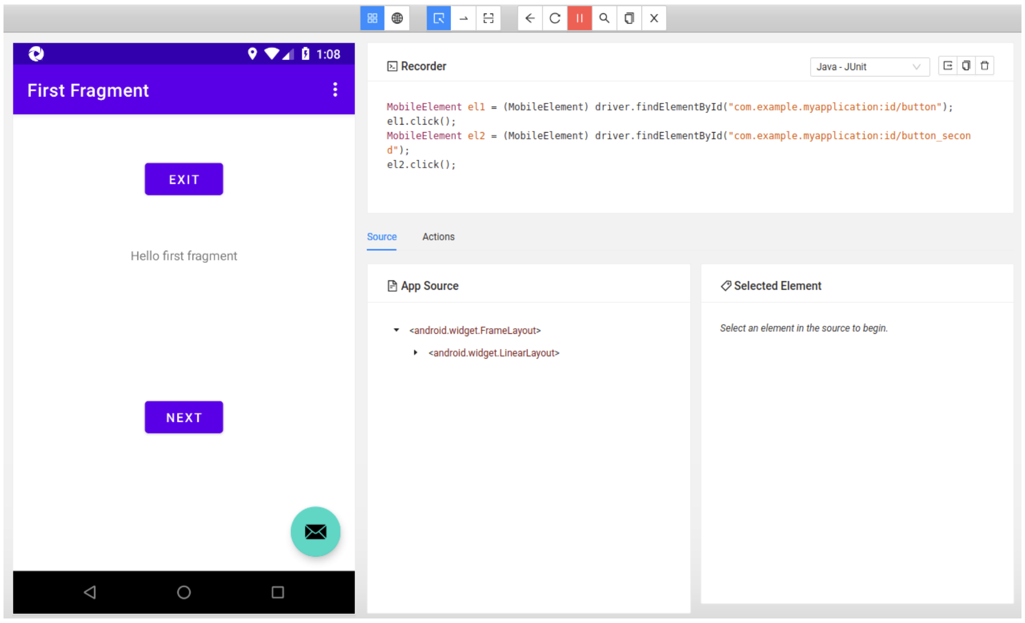
Appium Inspector[38] is a GUI inspector that allows a developer to perform test cases in GUI and inspect any element of mobile apps. It gives users an interface version of the Appium[30] so that we can construct test cases manually. This can also be used to identify the mobile app's UI elements. Appium-inspector does not support a web browser and is built for getting attributes of native apps specifically. The uses are as follows.

1. To record manual actions used on an app:- There is an option to record the test case actions. This project uses this feature to manually generate the code sequence related to failing test cases.

2. To find name, description, id, value and other attributes of elements/objects:- in appium inspector, we can use the UI to observe the characteristics of an element/object. Using this, we can directly find the element's id or the values of their attributes in the XML activity. This feature saves a lot of time; otherwise, we would need to dump the page source using appium directly. Also, this saves a lot of time for testers as we need to study the attributes exhaustively.

3. To understand element hierarchy:- by seeing the UI, we can understand how specific widgets/UI are aligned and different layers/fragments etc., the app may have. This may be trivial for developers but helpful info for testers.

The appium inspector is an appium client with a user interface. While the appium client does not support a GUI, the inspector allows us to select which server to use, which capabilities to set, and interact with the elements using GUI. We can see the XML structure of all visible parts on the screen. It provides us with a set of actions which can be performed and a recording feature. In this study, the record functionality of the Appium-Inspector was used to record the failing test case (a test case that crashes). In Figure 3.1, the recorder shows a snippet Appium-Inspector connected to an application.



```
MobileElement el1 = (MobileElement) driver.findElementById("com.example.myapplication:id/button");
el1.click();
MobileElement el2 = (MobileElement) driver.findElementById("com.example.myapplication:id/button_secon
d");
el2.click();
```

Figure 3.1: Appium-inspector to write Test case

3.4 Important Modules/Other Softwares

ADB[44] or Android Debug Bridge is a tool (command line) that helps us communicate with a device. It aids us in many operations regarding android devices, like installing and debugging apps.

Genymotion emulator any emulator (which supports root access) or real device can be used in this project to launch the mobile app. For testing genymotion emulator was used.

Dexdump[39] For an android app APK on extraction, we get a class.dex file. All Activity, Object, and Fragment used within the source code will be stored as bytes in the dex file. Dexdump creates a Dalvik Virtual Machine bytecode on running the tool on a dex file. The output of the dexdump is then formatted to get a list of classes or references.

Chapter 4

Algorithm and Implementation

4.1 Hooking and Pre-processing of AUT

Pre-processing of AUT takes place before we start running a test case. The Pre-processing includes:

- Getting user-defined class names in the source code of AUT.
- Getting user-defined method names in the source code of AUT.
- Hooking all user-defined methods.

The Pre-processing includes getting all class names because it is a requirement of Frida-js to get all methods. For getting all class names, we use *dexdump*[39], which is a disassembler tool. An android APK can be decompiled to get its dex files. These dex files are disassembled by *dexdump* to create the Dalvik Virtual Machine Bytecode. This Dalvik Virtual Machine code has a class descriptor field with the class names. Using this field, we get all user-defined class names.
Hooking an Android application and getting method names are done together(Algorithm 1).
Frida has an inbuilt message handler; this is a message handler that can pass messages between Frida-js/Android application and Frida-python. Frida-js is launched by Frida-python. After this Frida-python sends a list of classes to Frida-js using the message handler in Line 1 of algorithm 1. *Java.use* in Line 4 is a function which

Algorithm 1 Hooking AUT using Frida-js

```
1: $S \leftarrow listofclass$    ▷ get list of class from Frida-python using message handler
2: for class in S do
3:     if class is $hookable$ then
4:         $hook \leftarrow Java.use(class)$    ▷ Hook of the class
5:         $methods \leftarrow hook.get\_declared\_methods()$    ▷ getting list of methods
6:         for method in methods do
7:             $sendMessage('method', method)$    ▷ message handler
8:             $overLoads \leftarrow hook[method].overloads$    ▷ list of overloads
9:             for overload in overLoads do
10:                $overload.implementation =$ function(){
11:                $sendMessage('entry', method)$    ▷ message handler
12:                $ret = this[method].apply(this, arguments)$    ▷ run the method
13:                $sendMessage('exit', method)$    ▷ message handler
14:                return ret
15:                }
16:            end for
17:        end for
18:    end if
19: end for
```

takes a class name as input and outputs a function to hook the class or methods defined in that class. So in Line 4, we get a function to hook a *class* or methods defined in that class. In line 5 *get_declared_methods* gets a list of all the user-defined methods in a class. Overloads of a method are list of methods having the same name but different input arguments. In line 8 we get list of *overloads* of the *method*. In line [10-15], we change the implementation of an overload to execute the message handlers before and after the method is called in, line 12.

In summary, the method names are passed to Frida-python in Line 7 on every iteration; this way, the list of methods is generated. Also, all methods are iterated over class by class and for every method; all their overloads are hooked to send messages to Frida-python whenever the method is used in the AUT during a test case run. These messages are sent before and after the method is called, with their flags being 'entry' and 'exit', respectively.

4.2 Test case generation and Crash Handling

4.2.1 Test case generation(binary vector)

The Pre-processing leads to the app's instrumentation regarding which methods are run when any action is performed on AUT. The message passing takes place using Frida's inbuilt function called *on_message* a message handler. Four flags are used in messages: entry, exit, method, and classes. Their format and what they symbolise are explained below.

- Entry: 'entry' flag followed by its methods name. A message with this flag is used to indicate entry or signal the start of the corresponding method. Sent from AUT to Frida-python.
- Exit: 'exit' flag followed by its methods name. A message with this flag is used to indicate exit or signal the end of the corresponding method. Sent from AUT to Frida-python.
- Method: 'method' flag followed by its methods name. A message with this flag is used to send all method names before we run a test case. Sent from AUT to Frida-python.
- Classes: 'class' flag followed by a list of classes. A message with this flag is used to send the list of classes of the Android application. Sent Frida-python to Frida-js.

The algorithm 2 shows how the test case x of test suite X is generated. The method *on_message* described in line 4 of the algorithm 2 is part of Frida-python code. Whenever a message is received by Frida-python or is to be sent from Frida-python,*on_message* method is executed. Only part of the code where Frida-python sends data to Frida-js is sending the class list in line 6. This sends the list of classes of AUT, which was derived using dexdump[39]. Line [8-12] handles the initialisation of the test case and mapping method name to component number.

Algorithm 2 Generating TestCase

```
 1: $arr \leftarrow []$                                          ▷ test case
 2: $map \leftarrow \{\}$                  ▷ map method name to component number
 3: $MethodNumber \leftarrow 0$
 4: procedure ON_MESSAGE($message$)                         ▷ MessageHandler
 5:     if $message.type = "class"$ then
 6:         send('class',ClassList)          ▷ Sending list of classes to Frida-js
 7:     end if
 8:     if $message.type = "method"$ then
 9:         $map[message.payload] \leftarrow MethodNumber$
10:         $MethodNumber \leftarrow MethodNumber + 1$
11:         $arr.append(0)$
12:     end if
13:     if $message.type = "entry"$ then
14:         $arr[map[message.payload]] \leftarrow 2$
15:     end if
16:     if $message.type = "exit"$ and $arr[map[message.payload]] = 2$ then
17:         $arr[map[message.payload]] \leftarrow 1$      ▷ Means the method is exited
18:     end if
19: end procedure
20: $on\_message('class', ClassList)$
```

Line 9 maps the method name in the message's payload to a component number. Line 11 initialises the test case with 0 for each component(Methods). *If* condition in Line 13 handles the *'entry'* of any method, that is when a method is entered. The value of *2* is assigned for the component in the test case array(*arr*), but for Suspiciousness calculation in later sections; the value is taken as *1*. Similarly, the *if* condition on line 16 handles the *'exit'* of a method and the corresponding element on the test case array(*arr*) is set to *1*.
This is how messages are handled to generate a test case.

4.2.2 Crash Handling

Frida-python has an inbuilt method called *on_detached*, with return type boolean. When this method is called, it returns whether the present session(Frida launches a session at the start of a test case to hook the app) has ended or not. Using this method, the crash is instrumented in Frida-python. If the session is detached before Frida-python receives the signal of the end of the test case, then we conclude the app has crashed.

4.2.3 Finding actionable Widgets in an activity

Appium[30] has an inbuilt function for its driver called *getPageSource*, with return type *String*. This function returns the currently displayed activity in XML format. This string is then parsed using a java library to a DOM. Which is traversed to get the XPath of each widget. Every widget has the following attributes: package, class, text, resource-id, checkable, checked, clickable, enabled, focusable, focused, long-clickable, password, scrollable, bounds, and displayed.
Actionable widgets are differentiated from non-actionable based on the values of these attributes. Examples of criteria to select actionable widgets are:

- if a widget has an attribute clickable as true or checkable as true, then we can click the element.
- if a widget has class as ”Edit Text” and focusable attribute is true; then we can edit the text box.
- if a widget has scrollable as true, then we can scroll in the given bound(this is also an attribute).

4.3 Automated Test Generation

Algorithm 3 is the pseudo-code for automated test generation. It takes 3 inputs:

1. AUT
2. Test suite completion criteria
3. Failing test case.

Algorithm 3 Spectrum generation and automatic test generation

```
1: Input:Application under test $AUT$
2: Input:test suite completion criteria $c$
3: Input:List of actions of failing test case $F$
4: $T \leftarrow c.NumberOfTests$                          ▷ Number of tests to perform
5: $N \leftarrow c.TestCaseLength$                         ▷ Length of each test case
6: $Spectrum \leftarrow \phi$
7: for Action in F do                                     ▷ Failing test case is run
8:     Run(Action)
9: end for
10: $error \leftarrow checkForCrash()$                     ▷ 1 if error 0 otherWise
11: $t \leftarrow getTestCase()$                           ▷ Get test case form Frida-python
12: $s \leftarrow (t, error)$
13: $Spectrum \leftarrow Spectrum \cup s$
14: for $i \in (0, T)$ do
15:     $j \leftarrow 0$
16:     for $j \in (0, sizeOf(F) - i)$ do
17:         Run(F[j])                                      ▷ Run $i$-th action
18:     end for
19:     for $k \in (j, N)$ do
20:         $Xpaths \leftarrow XMLParse(AUT)$              ▷ Get list of widgets
21:         $Xpath \leftarrow Random(Xpaths)$
22:         $widget \leftarrow getElementByXpath(Xpath)$
23:         $RunAction(widget)$
24:         $error \leftarrow checkForError()$             ▷ 1 if error 0 otherWise
25:         if error then
26:             break
27:         end if
28:     end for
29:     $t \leftarrow getTestCase()$                       ▷ Get test case form Frida-python
30:     $s \leftarrow (t, error)$
31:     $Spectrum \leftarrow Spectrum \cup s$
32: end for
```

The criteria for test suite completion(the number of test cases and the length of each test case) are set in a JSON file. In lines 8 and 17 *Run*() function executes the corresponding GUI Action from the list of actions in failing test case(an input). Line [7-9] runs the failing test case. *checkForCrash()* in Line 10 is a boolean return method which return 1 if app crashed otherwise return 0. *getTestCase* in Line 11 talks with Frida-python and gets the binary vector test case x from Frida-python. In Line [10-13] spectrum is updated, and new spectra of failing test case are added. The function *XMLParse* in line 20 first gets the present activity from AUT in XML format and then parses it and outputs list of XPaths of different widgets. Line 22 *getElementByXpath* is an inbuilt function of appium it gets a widget based on Xpath. Line 23 *RunAction* runs the corresponding action on the widget. The action is selected based on the values of the widget's attributes as described in the Dynamic Analysis section 4.2.3. In Line 24, after every GUI action in line 23 it checks if the app crashes. If it crashes, then the loop of the present test case is exited and Spectrum is updated. The outer loop from lines [14-32] is run T times which is the number of test cases. The two inner loops on a whole run N times, which is the length of each test case.
After every test case run, the Spectrum is updated in line [29-31].

4.4 Spectrum Based Fault localisation(SBFL)

Spectrum S is the output of Algorithm 3 in section 4.3. Spectrum can be represented as $S = (A, e)$ 1.1 where A is the Activity matrix and e is the error vector. The columns of Activity matrix A are user-defined methods of AUT, and rows are the test cases. On this Spectrum S we use Ochiai[45] ranking metric4.1 to get the *Suspiciousness* Score of each component(method).

$$Ochiai_j = \frac{e_f}{\sqrt{(e_f + n_f)(e_f + e_p)}} \tag{4.1}$$

For j^{th} component where e_f is count of a_{ij} where a_{ij}=1 and e_i=1,e_p is count of a_{ij} where a_{ij}=1 and e_i=0 and n_f is count of e_i=1. In fig 1.1 a_{ij} refers to in i^{th} testcase whether j^{th} component was executed or not and error vector is positive for failing test case.

The components are then ranked in decreasing order of their Ochiai Score(Suspiciousness). The lower the rank of the buggy component, the better. The developer needs to examine fewer components before the bug is found. This effort is termed Cost of Diagnosis (D) and is measured as $\frac{r}{m}$ where r is the rank of a faulty component in m components and m is the number of members. The cost of diagnosis can also be defined by the Wasted Effort, which is the metric we will use in this project. Wasted Effort(W) measures the number of non-buggy components we examine before we examine a buggy component. It goes by $\frac{r-1}{m-1}$ where r is the rank of a buggy component in m components.

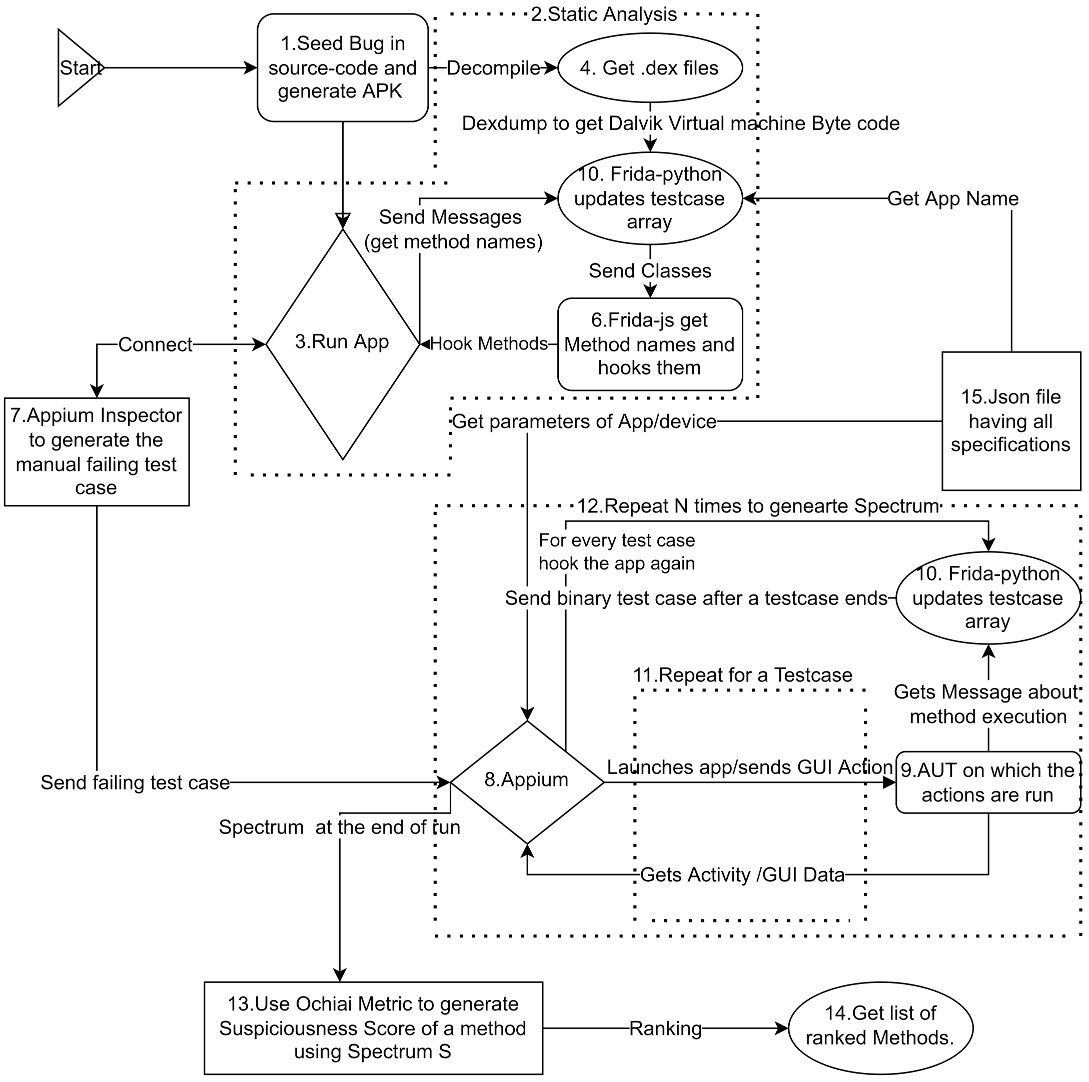
4.5 Summary

The experimentation on SpecDroid starts with bug seeding (seeding a bug in source code of AUT) as shown in box number 1 of flowchart 4.1. After seeding the bug, AUT is installed on the device. Then Pre-processing(dotted box 2) and hooking of the AUT are done. In the Pre-processing, we decompile the APK to get dex files as in box 4 then using dexdump we get classes, which are then input to Frida-python (box 10). These classes are then sent to Frida-js (box 6). Then the application is hooked by Frida-js and method names are sent from the application to Frida-python, which uses it to initialise the test case and map method names to component numbers.

Appium-Inspector is used for writing the failing test case. In (box 7), Appium-Inspector launches the AUT, and we manually write the failing test case. That failing test case is input to Appium main code.

In the JSON file (box 15), we store all the data required for the framework Spec-

Figure 4.1: Flow Chart[46] of the Framework SpecDroid for the experiments conducted


Start
1.Seed Bug in source-code and generate APK
2.Static Analysis
Decompile
4. Get .dex files
Dexdump to get Dalvik Virtual machine Byte code
10. Frida-python updates testcase array
Get App Name
Send Messages (get method names)
Send Classes
3.Run App
Hook Methods
6.Frida-js get Method names and hooks them
Connect
7.Appium Inspector to generate the manual failing test case
15.Json file having all specifications
Get parameters of App/device
12.Repeat N times to genearte Spectrum
For every test case hook the app again
Send binary test case after a testcase ends
10. Frida-python updates testcase array
11.Repeat for a Testcase
Gets Message about method execution
Send failing test case
8.Appium
Launches app/sends GUI Action
9.AUT on which the actions are run
Spectrum at the end of run
Gets Activity /GUI Data
13.Use Ochiai Metric to generate Suspiciousness Score of a method using Spectrum S
Ranking
14.Get list of ranked Methods.

Droid to run. Some of these specifications are the number of test cases, length of each test case, name of the AUT, device specification, etc. This JSON file is accessed by Appium (arrow box 15 to box 8) to get test completion criteria. This is also accessed by Frida-python (arrow box 15 to box 10) to get $AppName$(name of application). The $AppName$ is used by Frida-python to connect to the AUT and hook it before the test case is run.

Dotted box 12 includes all steps involved in Spectrum generation. The main code in box 8 Appium runs all the test cases based on algorithm 3. The dotted box 11 shows steps undertaken to generate a test case. The test case is run action-wise. Whenever an action is sent to be run on AUT (box 9), the message handler is called by the methods which are run; this data is sent to Frida-python (box 10). This happens throughout the run of a test case, and at the end of the test case, Appium receives the binary array from Frida-python (arrow box 10 to box 8). Also, we check for crashes after every action (arrow box 9 to box 8). If a crash occurs, the test case is terminated early. The spectra corresponding to a test case has 1 as its error bit if a crash occurs or its 0.

This process is run N times (Number of test cases) to generate the spectrum. Ochiai metric is run on every component in box 13 to get their suspiciousness score. Then the rank of each method is output in box 14. This is output in a CSV file along with the spectrum.

Chapter 5

Experiments

Each test suite X is evaluated based on Cost of Diagnosability D using Wasted effort W.

In this experimentation, we pose two research questions.

RQ1 How does SpecDroid perform in terms of Cost of Diagnosability D in finding the seeded bugs over our benchmark apps?

RQ2 How does the Cost of Diagnosability D vary with the number of runs?

All Android Apps and their source code for seeding were taken from Fdroid website[47] in the experimentation. Fdroid is a store for Android Apps and a repository for verified free software Applications. Seven apps were chosen randomly from Fdroid, with the criteria that they were built using either Android Java or Kotlin.

5.1 Bug seeding

Bug seeding here refers to manually introducing exceptions at specific locations in the source code of an Android application. So that when the exception is executed the application will crash.

Crash doesn't occur for every user interaction with a certain widget which calls a buggy method. It crashes for some interactions, and other times, it won't. For example:- In a form with many text boxes, when we click the Submit button, a crash may occur, but this crash may be unrelated to the button and may depend on the input of a text box. To mimic this behaviour, we seed bugs based on algorithm 4.

The *OtherCode* in lines [3,6,8,11,15] represent some other code written by the

Algorithm 4 Method

```
 1: procedure MainClass
 2:     $x = 10$
 3:     Other Code
 4: end procedure
 5: procedure Method1($a$)
 6:     Other Code
 7:     $x = 5$
 8:     Other Code
 9: end procedure
10: procedure Method2($b$)                              ▷ Crash happens here
11:     Other Code
12:     if $x = 5$ then
13:         Crash
14:     end if
15:     Other Code
16: end procedure
```

developer. In line [1-4] of algorithm 4 a variable x is initialised to 10. In line [4-9] method $Method1$ of some class, the variable is reassigned to 5. In line [10-15] method $Method2$ of some class, there is an If condition on line 12 which is satisfied if the variable x is 5. If the condition is satisfied, then the app crashes. So this algorithm shows if the $Method1$ is called before the execution of $Method2$ then a crash occurs on the execution of $Method2$. This kind of bug follows a "$cause$" and "$effect$" relation.

These kinds of bugs were seeded manually in the 7 apps. There were two versions of seeded bugs. So a total of 14 seeded apps were formed. With each app having one bug.

5.2 Experiments and Results on Top-k and Wasted Effort metrics(RQ1)

Fourteen seeded applications were used for the experiments. SpecDroid was run three times on every app, with each run having 11 test cases and each test case

AppName	Number of Method	Version 1			Version 2		
		1	2	3	1	2	3
Score sheets	101	1	1	1	13	1	5
Mines3D	197	1	1	1	12	6	7
OneTwo	156	5	1	5	1	1	1
Quick Calculation	83	9	9	9	9	10	9
Tabletop Tools	137	1	1	1	1	1	1
WordleSolver	33	1	1	1	1	1	1
Rental Calc	145	1	1	1	1	1	1

Table 5.1: Minimum Ranks of buggy component for every run

AppName	Number of Method	Version 1			Version 2		
		1	2	3	1	2	3
Score sheets	101	1	1	1	18	2	12
Mines3D	197	1	1	1	13	10	9
OneTwo	156	7	15	9	3	2	2
Quick Calculation	83	10	10	10	10	13	22
Tabletop Tools	137	3	3	3	1	1	1
WordleSolver	33	1	1	1	2	2	2
Rental Calc	145	2	14	2	2	2	2

Table 5.2: Maximum Ranks of buggy component for every run

having 20 GUI actions except the first test case, which is manually input. The data regarding the rank of the buggy component in every run is listed in table 5.1. We are using two Methods(”cause” and ”effect”, as mentioned in the above section) in seeding the bug. Hence the minimum of the ranks of both methods is taken as the rank of the buggy component.

In every run all the components get Ochiai score, many components may get the same score so the analysis was done based one two ranks. One is its maximum possible rank and one is the minimum possible rank. The ranks in both the case are listed below.

5.2.1 Wasted effort

Using the data from table 5.1 and 5.2, we calculate wasted effort for each app in table 5.3. This wasted effort is the median of all six runs(3 from each version). The median of maximum wasted effort W over all bugs is 0.011. This means if we follow the rank generated by SpecDroid we are likely to localise the bug by observing

1.1% of methods of our given list of ranks.

App Name	Minimum Wasted effort	Maximum Wasted effort
Score sheets	0.02	0.055
Mines3D	0.015	0.023
OneTwo	0.013	0.029
Quick Calculation	0.097	0.128
Tabletop Tools	0.0	0.007
WordleSolver	0.0	0.0156
Rental Calc	0.0	0.007

Table 5.3: Wasted effort

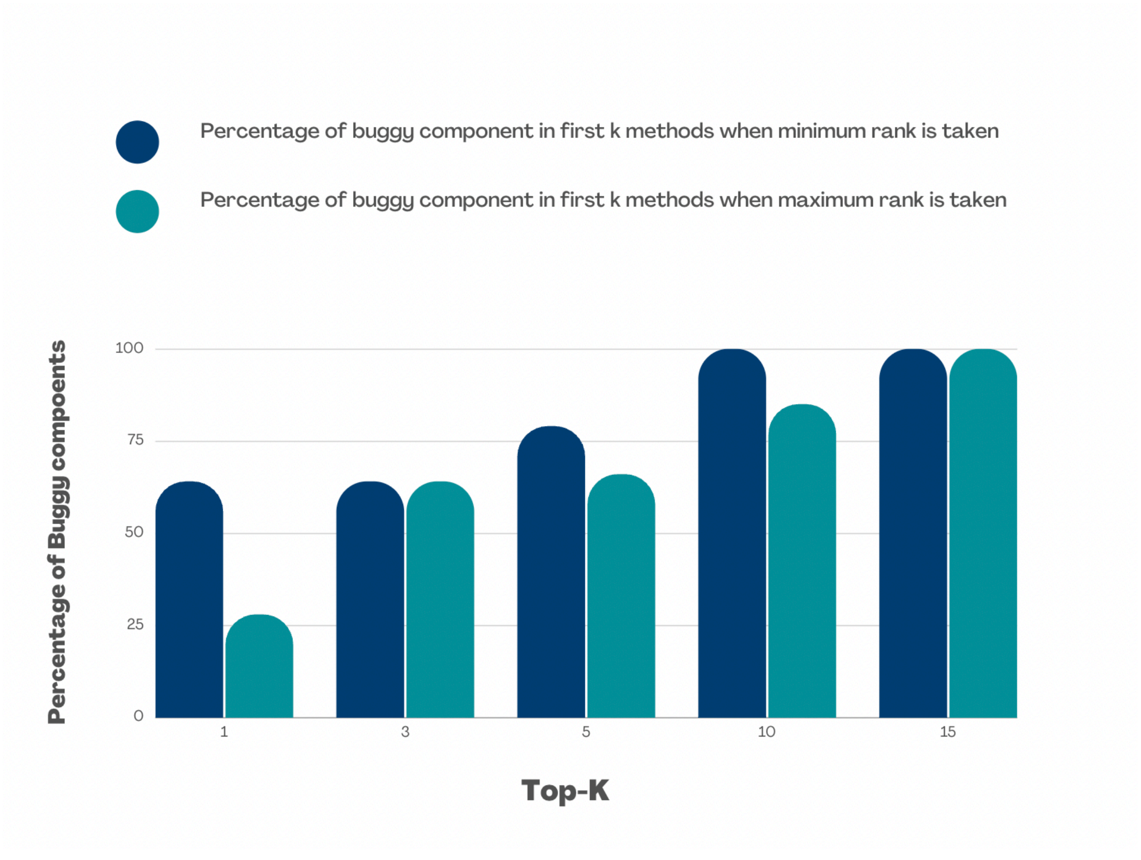
5.2.2 Top-k

Top-k refers to the ”Number of buggy methods having their rank less than k”.

In bar-graph 5.1 we have considered top 1,3,5,10 and 15 to draw the bar graph.

We can observe that most apps have ranks in top 3, i.e. there are 10 out of 14 apps

Figure 5.1: Graph of Buggy methods on Top-k Metric for Minimum Rank and Maximum Rank



have their bugs found just by observing top three methods, and all apps have their buggy methods found in 15 ranks.

5.3 Cost of Diagnosis and number of test cases(RQ2)

Five out of 14 buggy versions of app was chosen to perform this experiment. The number of the test case T was changed and cost D was observed. We conducted experiments on 3 T. For [5,10,20]. The result of each run is in the table 5.4,5.5.

AppName	T=5			T=10			T=20		
	1	2	3	1	2	3	1	2	3
Score sheets	7	6	5	13	1	5	3	4	6
Quick Calculation	1	9	9	9	9	9	9	9	9
Mines3D	1	3	5	12	6	7	65	10	7
OneTwo	24	5	1	5	1	5	1	5	1
Rental Calc	1	1	1	1	1	1	1	1	1

Table 5.4: Minimum ranks when changing number of test case per run

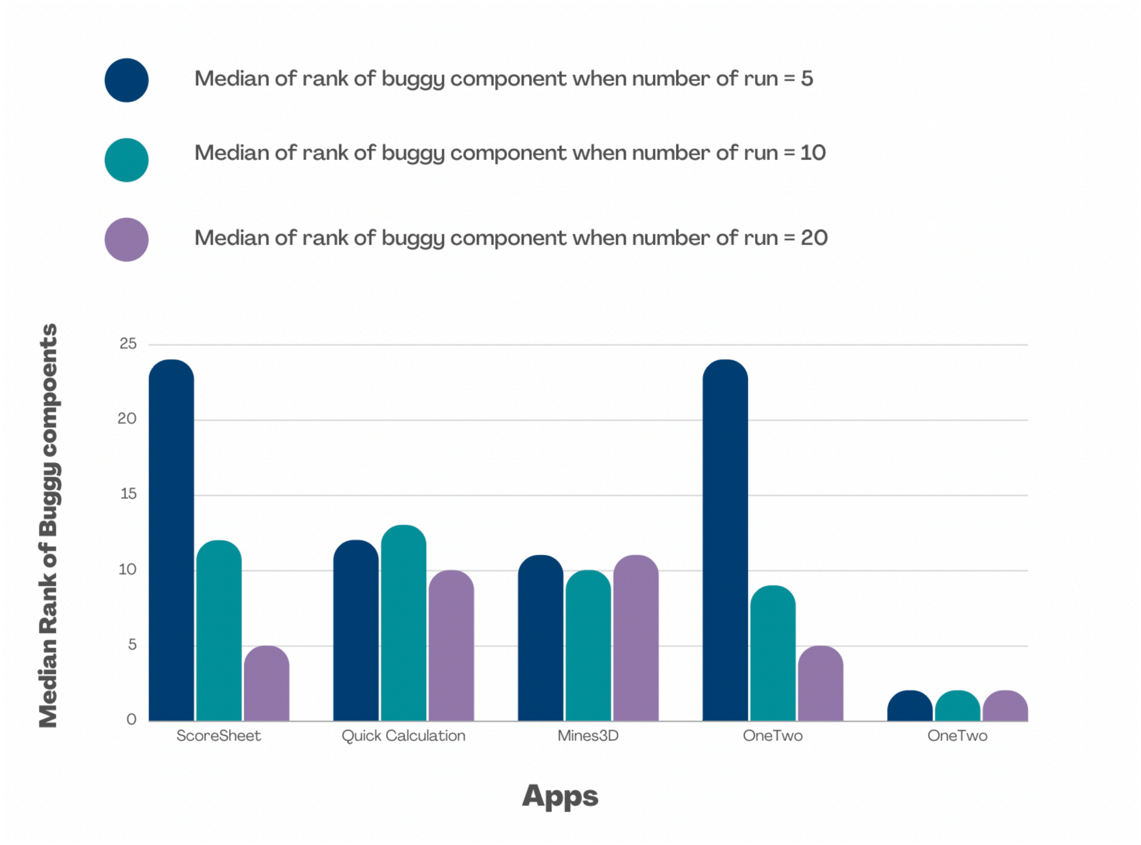
AppName	T=5			T=10			T=20		
	1	2	3	1	2	3	1	2	3
Score sheets	24	24	24	18	2	12	3	5	12
Quick Calculation	20	12	12	10	13	22	10	22	10
Mines3D	2	11	11	13	10	9	67	11	8
OneTwo	39	16	24	7	15	9	5	5	5
Rental Calc	2	2	2	2	14	2	2	2	2

Table 5.5: Maximum ranks when changing number of test cases per run

The median of the rank of buggy components for all apps in the maximum rank table for $T = 5$ is 12 which decreases to 10 for $T = 10$ which further decreases to 5 for $T = 20$. Also, the effort W needed to find the buggy component decreased on increasing the number of test cases per run.

This can further be seen in the table 5.2 which shows the median rank of buggy components for every app and runs.

Figure 5.2: Rank of buggy components for every app



We can conclude that the result improves when we increase the number of runs. This is because as we increase T, the Coverage of the AUT is supposed to increase. Every widget in an activity has an equal probability of execution, and if more widgets are acted upon, more methods are executed. In this experiment on increasing the number of runs, the Wasted Effort to find the buggy method was reduced.

Chapter 6

Conclusions

We propose framework SpecDroid which uses Random testing along with SBFL and is built on Appium[30] framework for Debugging Android applications. This framework uses Frida[37] to hook the application, and we get coverage data dynamically. Hooking to get coverage data is not popular, but in this project, we conclude hooking can be a viable method to get coverage data dynamically. Each of the runs on seeded apps took around 8-9 minutes. A run is for 10 test cases having 20 Actions and one user-defined test case. SBFL is also rarely used in Android app testing as the metrics usually used to measure the tool's efficiency are coverage and the number of crashes. SBFL ranks methods based on their suspiciousness scores. In the experiments conducted, SBFL performed well on the benchmark apps selected from Fdroid as the median of Wasted Effort W on all runs is low.

6.1 Future work

Good Testcase(based on Coverage or DDU Score of the Test Suite[48])is a must for SBFL to give good results, but we are only using Random-testing with a basic algorithm and also using manually generated failing Testcase. Also, the dynamic coverage data which we get from Frida is underutilised as we are doing random testing. Introducing an intelligent test data generation using techniques like Q-learning, Fuzzing, etc will surely improve the tool to get better test suites.

References

[1] *Mobile Operating System Market Share Worldwide.* `https://gs.statcounter.com/os-market-share/mobile/worldwide`. [Online; accessed 25-June-2022].

[2] *Number of available applications in the Google Play Store from December 2009 to March 2022.* `https://buildfire.com/app-statistics/`. [Online; accessed 25-June-2022].

[3] *Top 7 Mobile App Failures Learnings to Build a Successful App.* `https://appinventiv.com/blog/mobile-app-failures-and-learnings-to-build-a-successful-app/#:~:text=According%20to%20Statista%2C%20the%20global,in%20the%20in%2Dapp%20purchases.`. [Online; accessed 25-June-2022].

[4] *2022 Cell Phone Usage Statistics: How Obsessed Are We?* `https://www.reviews.org/mobile/cell-phone-addiction/`. [Online; accessed 25-June-2022].

[5] *Number of Android apps on Google Play.* `https://www.appbrain.com/stats/number-of-android-apps`. [Online; accessed 25-June-2022].

[6] Hammad Khalid et al. "What Do Mobile App Users Complain About?" In: *IEEE Software* 32.3 (2015), pp. 70–77. DOI: `10.1109/MS.2014.50`.

[7] *FAILING TO MEET MOBILE APP USER EXPECTATIONS.* `https://techbeacon.com/sites/default/files/gated_asset/mobile-app-user-survey-failing-meet-user-expectations.pdf`. [Online; accessed 25-June-2022].

[8] Husam N. Yasin, Siti Hafizah Ab Hamid, and Raja Jamilah Raja Yusof. "DroidbotX: Test Case Generation Tool for Android Applications Using Q-Learning". In: *Symmetry* 13.2 (2021). ISSN: 2073-8994. DOI: `10.3390/sym13020310`. URL: `https://www.mdpi.com/2073-8994/13/2/310`.

[9] Domenico Amalfitano et al. "Using GUI Ripping for Automated Testing of Android Applications". In: *ASE '12: Proceedings of the 27th IEEE international conference on Automated software engineering.* Washington, DC, USA: IEEE Computer Society, 2012.

[10] Ting Su et al. "Guided, stochastic model-based GUI testing of Android apps". In: Aug. 2017, pp. 245–256. DOI: `10.1145/3106237.3106298`.

[11] 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.* 2016, pp. 94–105.

[12] *UI/Application Exerciser Monkey|Android Developers.* `https://developer.android.com/studio/test/monkey`. [Online; accessed 16-July-2022].

[13] Stefan Karlsson et al. “Model-based Automated Testing of Mobile Applications: An Industrial Case Study”. In: *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2021, pp. 130–137. DOI: 10.1109/ICSTW52544.2021.00033.

[14] aptimizely. *Mobile app A/B testing*. https://www.optimizely.com/optimization-glossary/mobile-app-ab-testing/. [Online; accessed 31-July-2022].

[15] Rui Abreu et al. “A practical evaluation of spectrum-based fault localization”. In: *Journal of Systems and Software* 82 (Nov. 2009), pp. 1780–1792. DOI: 10.1016/j.jss.2009.06.035.

[16] Spencer Pearson et al. “Evaluating and Improving Fault Localization”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), pp. 609–620.

[17] Pingfan Kong et al. “Automated Testing of Android Apps: A Systematic Literature Review”. In: *IEEE Transactions on Reliability* 68.1 (2019), pp. 45–66. DOI: 10.1109/TR.2018.2865733.

[18] Saswat Anand et al. “Automated Concolic Testing of Smartphone Apps”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, North Carolina: Association for Computing Machinery, 2012. ISBN: 9781450316149. DOI: 10.1145/2393596.2393666. URL: https://doi.org/10.1145/2393596.2393666.

[19] Mario Linares Vásquez et al. “Enabling Mutation Testing for Android Apps”. In: *CoRR* abs/1707.09038 (2017). arXiv: 1707.09038. URL: http://arxiv.org/abs/1707.09038.

[20] Wasif Afzal, R. Torkar, and Robert Feldt. “A systematic review of search-based testing for non-functional system properties”. In: *Inf. Softw. Technol.* 51 (2009), pp. 957–976.

[21] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. “EvoDroid: Segmented Evolutionary Testing of Android Apps”. In: FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014. ISBN: 9781450330565. DOI: 10.1145/2635868.2635896. URL: https://doi.org/10.1145/2635868.2635896.

[22] 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. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 94–105. ISBN: 9781450343909. DOI: 10.1145/2931037.2931054. URL: https://doi.org/10.1145/2931037.2931054.

[23] Dhanoop Karunakaran. *The Actor-Critic Reinforcement Learning algorithm*. [Online; accessed 31-July-2022]. 2020. URL: https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14.

[24] Wikipedia contributors. *Deep Q-Networks*. [Online; accessed 31-July-2022]. 2022. URL: https://www.techopedia.com/definition/34032/deep-q-networks.

[25] Wikipedia contributors. *State–action–reward–state–action*. [Online; accessed 31-July-2022]. 2022. URL: https://en.wikipedia.org/wiki/State%5C%E2%5C%80%5C%93action%5C%E2%5C%80%5C%93reward%5C%E2%5C%80%5C%93state%5C%E2%5C%80%5C%93action.

[26] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: An input generation system for Android apps". In: Aug. 2013, pp. 224–234. DOI: 10.1145/2491411.2491450.

[27] Kun Yang et al. "IntentFuzzer: Detecting Capability Leaks of Android Applications". In: ASIA CCS '14. Kyoto, Japan: Association for Computing Machinery, 2014, pp. 531–536. ISBN: 9781450328005. DOI: 10.1145/2590296.2590316. URL: https://doi.org/10.1145/2590296.2590316.

[28] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. "EvoDroid: segmented evolutionary testing of Android apps". In: Nov. 2014, pp. 599–609. DOI: 10.1145/2635868.2635896.

[29] Cong Zheng et al. "Smartdroid: an automatic system for revealing uibased trigger conditions in android applications". In: *In Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices (2012), ACM*, pp. 93–104.

[30] *Appium: Mobile app automation made awesome.* http://appium.io/. [Online; accessed 10-April-2022].

[31] *Mobile app A/B testing.* https://android-doc.github.io/tools/help/uiautomator/index.html. [Online; accessed 31-July-2022].

[32] Tanzirul Azim. *A3E*. [Online; accessed 31-July-2022]. URL: http://www.tanzirul.com/research/a3e.

[33] Tianxiao Gu et al. "AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 103–114. DOI: 10.1109/ICSME.2017.72.

[34] Thi Anh Tuyet Vuong and Shingo Takada. "A Reinforcement Learning Based Approach to Automated Testing of Android Applications". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 31–37. ISBN: 9781450360531. URL: https://doi.org/10.1145/3278186.3278191.

[35] David Adamo et al. "Reinforcement Learning for Android GUI Testing". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 2–8. ISBN: 9781450360531. URL: https://doi.org/10.1145/3278186.3278187.

[36] Ting Su et al. "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. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 245–256. ISBN: 9781450351058. DOI: 10.1145/3106237.3106298. URL: https://doi.org/10.1145/3106237.3106298.

[37] *Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.* `https://frida.re/`. [Online; accessed 10-May-2022].

[38] 12 others. jlipps. *Appium-inspector.* `https://github.com/appium/appium-inspector`. 2022.

[39] Lars Vogel. *Disassemble Android dex files.* [Online; accessed 31-July-2022]. 2011. URL: `http://blog.vogella.com/2011/02/14/disassemble-android-dex/`.

[40] *Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.* `https://frida.re/docs/frida-ps/`. [Online; accessed 10-May-2022].

[41] *Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.* `https://frida.re/docs/frida-trace/`. [Online; accessed 10-May-2022].

[42] *Robotium - Open Source Testing Android User Interface.* `https://www.methodsandtools.com/tools/robotium.php`. [Online; accessed 31-July-2022].

[43] *Android Automation Testing with Ranorex.* `https://www.ranorex.com/mobile-automation-testing/android-test-automation/`. [Online; accessed 31-July-2022].

[44] Android developer. *Android Debug Bridge (adb).* URL: `https://developer.android.com/studio/command-line/adb`.

[45] Aitor Arrieta et al. “Spectrum-based fault localization in software product lines”. In: *Information and Software Technology* 100 (2018), pp. 18–31. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2018.03.008`. URL: `https://www.sciencedirect.com/science/article/pii/S095058491730188X`.

[46] Prajwal HG. *FlowChart.* [Online; accessed 31-July-2022]. URL: `https://drive.google.com/file/d/1xHqVUaxsYiJWNPx6V8c7bBmzTOHeR_Ja/view?usp=sharing`.

[47] *F-Droid is an installable catalogue of FOSS (Free and Open Source Software) applications for the Android platform. The client makes it easy to browse, install, and keep track of updates on your device.* `https://f-droid.org/`. [Online; accessed 24-July-2022].

[48] Alexandre Perez, Rui Abreu, and Arie van Deursen. “A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* 2017, pp. 654–664. DOI: `10.1109/ICSE.2017.66`.

[49] Albert Einstein. “Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]”. In: *Annalen der Physik* 322.10 (1905), pp. 891–921. DOI: `http://dx.doi.org/10.1002/andp.19053221004`.

[50] Paul Adrien Maurice Dirac. *The Principles of Quantum Mechanics.* International series of monographs on physics. Clarendon Press, 1981. ISBN: 9780198520115.

[51] Donald E. Knuth. “Fundamental Algorithms”. In: Addison-Wesley, 1973. Chap. 1.2.

[52] Nariman Mirzaei et al. “SIG-Droid: Automated system input generation for Android applications”. In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 2015, pp. 461–471. DOI: `10.1109/ISSRE.2015.7381839`.

[53] 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. Hong Kong, China: Association for Computing Machinery, 2014, pp. 599–609. ISBN: 9781450330565. DOI: `10.1145/2635868.2635896`. URL: `https://doi.org/10.1145/2635868.2635896`.

[54] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. “Systematic Execution of Android Test Suites in Adverse Conditions”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 83–93. ISBN: 9781450336208. DOI: `10.1145/2771783.2771786`. URL: `https://doi.org/10.1145/2771783.2771786`.

[55] Yury Zhauniarovich et al. “Towards Black Box Testing of Android Apps”. In: *2015 10th International Conference on Availability, Reliability and Security* (2015), pp. 501–510.