AUTOMATED TESTING TO DETECT STATUS DATA LOSS IN ANDROID APPLICATIONS

A Thesis by

Anusha Konduru

Bachelor of Technology, Narayana Engineering College, 2010

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfilment of
the requirements for the degree of
Master of Science

May 2020

AUTOMATED TESTING TO DETECT STATUS DATA LOSS IN ANDROID APPLICATIONS

The following faculty members have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfilment of the requirement for the degree of Master of Science, with a major in Computer Science.

Zhiyong Shan, Committee Chair

Eylem Asmatulu, Committee Member

Vinod Namboodiri, Committee Member

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my research advisor, Zhiyong Shan, for his guidance and support in this research. I would also thank my senior, Hoang Dang for helping me in understanding the tools used in the research. I extend my gratitude to the members of my committee, Eylem Asmatulu and Vinod Namboodiri for their valuable comments and suggestions on this research.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF ABBREVATIONS

APK	Android Package Kit
SDK	Software Development Kit
App	Application
UI	User Interface
IDE	Integrated Development Environment
GUI	Graphical User Interface
CPU	Central Processing Unit
TCP	Transmission Control Protocol
API	Application Programming Interface
USB	Universal Serial Bus

ABSTRACT

Mobile applications are increasing in a significant amount, each to address the requirements of many users. However, the quick developments and enhancements are resulting in many underlying defects. When user installs an app, it is expected that the data should be persistent at all states, which requires saving the instance data for all the activities. An app can be paused or sent to the background due to other interruptions or user intervention. It is difficult for the programmer to test this issue for all the activities. This results the issue of data loss, the information/details entered by user in the app are not saved when there is any interruption. This results in degrading the user experience because user should enter the information each time there is an interruption. There are several static analysis tools to test this issue. Thus, the automated testing to detect such data loss is important to improve the user experience. This research proposes a tool, DroidDLP, a Data Loss Preventor in Android, which will detect the information loss from a given android application. We have tested 395 applications and found 12 applications with the issue of data loss. This approach is proved highly accurate and reliable to find the apps with this defect, which can be used by android developers to avoid such errors.

CHAPTER 1

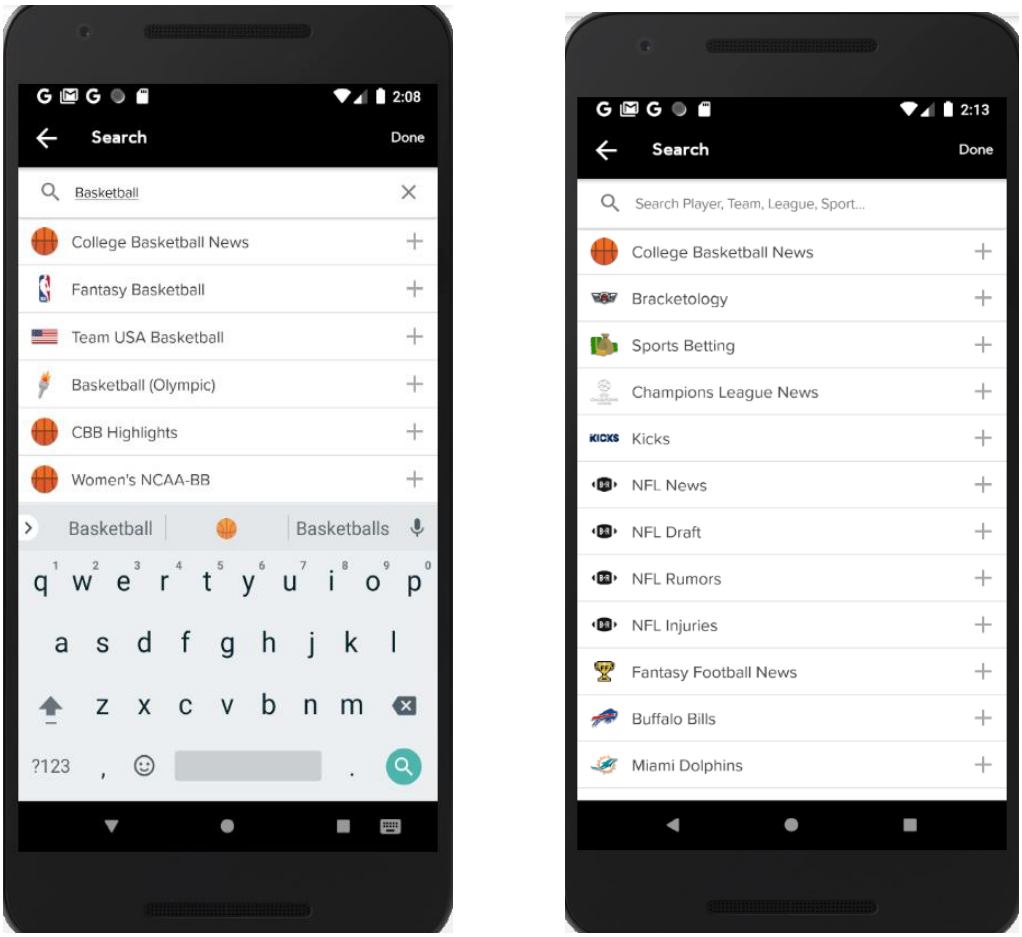
INTRODUCTION

Android platform offers the support to spare the data identified with the application. A large portion of the engineers utilize this feature during the design and development of app, to maintain persistent data for better client experience. When there is a user-initiated or framework-initiated event that sends the application to the platform, the application should deal with the save state condition to store the data. For instance, when a client opens a shopping application and quest for an item in the search bar and afterward presses back/home button or answers a call, the client expects the search results to be accessible after the action. Thus, the developers are capable to deal with such cases by saving the state and the data before switching to other window. While a large portion of the developers follow this convention to build their applications, a portion of the applications are yet missing this, since this is certainly not a compulsory check for all the applications. Subsequently, the greater part of the applications is losing the information during activity change, like the user details entered during Signup, results in search action, checkbox selected.

To address this issue, an automated dynamic tool is intended to scan for inconsistent information in the applications. Rather than checking for the source code of the android application to check the state is spared, we utilize the UI to explore the app, starting with one activity then onto the next and observe the data. Android APK(Android Package Kit) is the essential input for this tool, which is utilized to install the app in the mobile device or the emulator. Likewise, Android SDK, Appium and Junit are utilized to create and test DroidDLP tool.

1.1 Motivational Example:

There are several types of errors when it comes to mobile applications. Though most of these bugs are found during the static analysis phase while designing or testing the apps. There are many errors which developers overlook while developing an application. Some of such errors are related to UI, some might be specific to performance and many others. When an application is run in the background it might consume a lot of space and CPU utilization, this leads to performance degradation. This can be identified by the existing generic tools to measure performance of any application in Android. But when it comes to the UI, there are many applications that are being added to the Google Play store every day. It is always difficult to design test cases for every single task for the app because most of them will be generic. For example, when an application is run and sent to background, it is expected to be on the same activity without any data loss when it is resumed. But in most of the applications this is not the case. There are many scenarios like the application might start over again from the main activity which is home page, the state of the activity will be constant but the data entered by the user will be lost which can be a result of search operation or the message typed by a user, or the data entered by a user during registration. For instance, user enters the word ‘Basketball’ to pick the teams who plays Basketball using an app called Bleacher Report. As mentioned in figure 1, we can see the results before sending the app to the background where user enters an input search value and gets a bunch of search results. After changing the UI state, that is sending the app to background and when the app resumes to the foreground, the search text entered by the user is no longer visible and the search results are lost as shown in figure 1.



a) App before Pause state b) App after Resume state

Figure 1: Example for Data Loss

Although it is possible to rerun the application and enter the data again, this will degrade the overall user experience and consume a lot of time depending on the rate of interruptions that user gets while performing the activity. Some of the interruptions might include getting an audio/video call or a notification from another application which sends the current application to background. This research introduces a tool to detect such issues irrespective of the app. User can select any app and give it as an input to the tool by placing the APK file with respect to the app in the project and start the test. DroidDLP tool performs a set of tests while navigating from one activity to other by sending the application to the background and resuming it.

1.2 Background:

Our research is based on Android, including Android system, Android applications, and Android application stores. This chapter will introduce shortly about relative information and the technology used in our research.

Android APK. An APK is the file format for applications utilized on the Android framework. APK files are accumulated utilizing the Android Studio, which is the integrated development environment (IDE) for building Android applications. An APK file incorporates all the product program's code and resources used to build up an application. Engineers who make applications for use on Android gadgets must assemble their application into the APK design preceding transferring to Google Play, the official marketplace for Android applications. APK files can likewise be circulated legitimately to other Android clients for installation on their devices, for example, emulators. Android clients can give authorization to their device to install unknown applications if they wish to download APK files and install them directly.

Android Life cycle. Whenever an app is opened, it is expected to perform some operations before getting exited or destroyed. These operations are called the activities which has a specific lifecycle called Android Life cycle. This research paper will use the OnPause() and OnStop() states to test whether there is data loss once the app is paused/killed and resumed. When an app is paused, stopped, or killed, data loss may occur[5]. We now proceed to define several restart levels in Android as shown in the fig , based on the amount of app state that is affected by restart; each restart level has one or more associated callback methods, automatically invoked by the Android Framework. We first discuss the levels and then describe the restarts' impact on data.

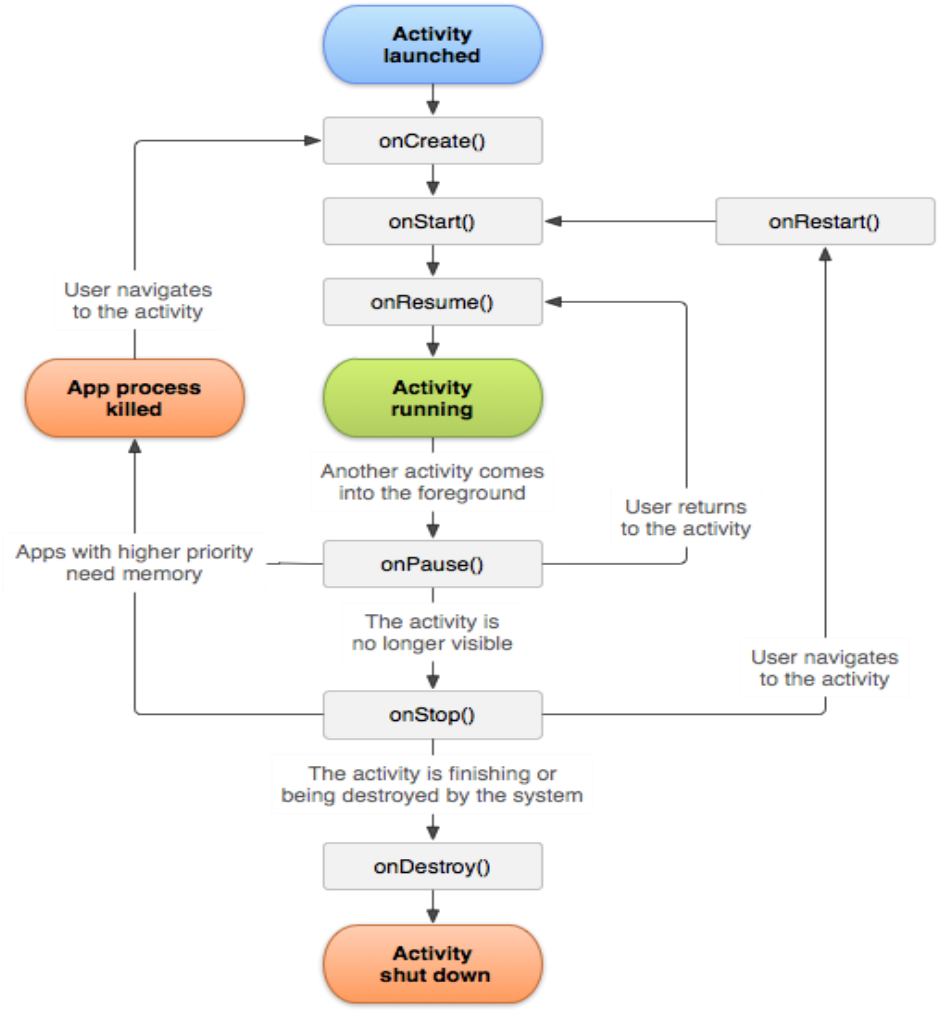


Figure 2: Android Life cycle

1. Pause State: When an activity is said to be in the Paused State, it implies that it is still running in the background such as a process behind another app of a popup or a transparent window which will make it invisible to the user on the phone screen. Android framework will not usually kill such activity in this state, but only in extreme case of resource crunch.
2. Stop State: When the activity is said to be in the Stopped state, the activity will no longer be visibile. For instance, when an activity is launched in the instance covers the entire screen and utilizes all the resources. Android framework also calls onStop(), when the activity has finished running and is about to be terminated.

3. Destroy State: A stopped activity is destroyed and all its data will be deleted when the Android Framework invokes onDestroy(). This can happen when the app is killed, or when user clicks the 'Back' button.

UIAutomator. There are many testing frameworks available to test the android applications. In this paper, Appium is used as the testing framework. Appium on Android uses the UIAutomator framework for automation[7]. UIAutomator is a framework built by android for automation purposes. Appium client accepts a program written in any language like c, Java, Python. Appium server connects with UIAutomator using JSON Wire Protocol. Thus creating an automation session for the mobile device and check if the desired capabilities are matching with the device. UIAutomator will interface with bootstrap.jar which is running in emulator/cell phone for performing user entered tasks bootstrap.jar acts as a TCP server, which sends instructions to perform the operations on the client(mobile device) with the help of UIAutomator.

The Appium android architecture diagram below gives a visual representation of the above steps.

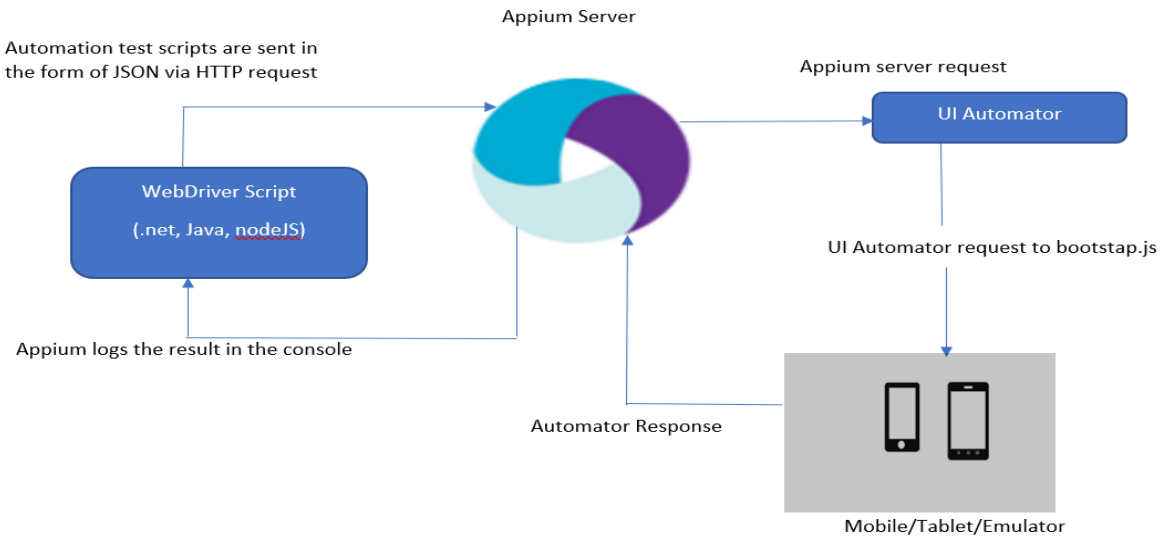


Figure 3: Appium server and UIAutomator

CHAPTER 2

LITERATURE REVIEW

Software testing can be done in several ways to find the failures and increase the performance of the software application. However, to add a testing tool for mobile testing many challenges are faced due to the different technologies and frameworks used to design and develop the app. UI testing adds new challenges requiring different and specialized new testing techniques for testing mobile apps. The main building blocks of Android applications are activities, services and content providers. Activities are nothing but the individual GUI views, whereas services are the processes that will be run to take of background processing, the receivers will listen to notifications initiated by the system (e.g. battery low, call) and finally the content providers share application data between apps. The Android GUI is primarily controlled using a touch screen, like most modern smartphone devices. The testing tools use this GUI by pressing the keys to navigate from one activity to other based on the scenario. There are different Mobile testing tools that can be classified according to different criteria of testing approaches.

2.1 Mutation testing:

When there is an error in the application, these tools provide set of mutation operations that can be performed on the app to avoid the error. A developer can use those operators in the next version of APK and test the application to check if the error is mitigated or not. Ana C. R. Paiva et. Al., created iMPAcT tool that automates the testing of recurrent behavior (UI patterns) present on Android applications[1]. This tool has presented 3 mutation operators created to mimic problems related to inadequate preservation of transient UI state when an app goes to background and returns to foreground. The aim of using the iMPAcT tool in this experiment is to check if the background & foreground test strategy defined and used by the tool can find the problems related to not

following the guidelines for implementing this behavior. If the test suite produced by the tool kills the mutants, we can suppose that it is complete regarding the testing of background & foreground transitions. Otherwise, it is incomplete. Consequently, Linares-Vasquez et. Al., [2], [7] created a taxonomy of Android bugs with the purpose of defining a list of 38 mutation operators for Android apps. Those operators were implemented in a tool called Mdroid +, which performs the mutations at source code level. Deng et al. [9], [10] also presented a set of 8 Android specific mutation operators; the mutations are oriented to change core components of Android apps (e.g., intents, event handlers, XML files and activity lifecycle). Additionally, Luna et al. [11] presented Edroid, a tool that uses 10 mutation operators oriented to validate changes in the GUI. Though the mutation tools work in most of the scenarios and improves the testing scope, this is purely to give suggestions to the developer, and this checks only transient UI state of the application. Our approach is fully automatic and will focus not only on the transient UI state but also the data entered by the user. Mutation tools usually work on the APK source code to parse and fetch the errors whereas our approach is to use the UIAutomator to get the UI elements of the app that is running on the device after deployment. This not only helps in find the errors but also the developer can retest the application using the DroidDLP tool to check if the error is resolved.

2.2 Static Analysis:

One of the effective ways to test a software application is using the static analysis tools, most of them being used before the program is run. Static analysis is used by the developers to find the bugs in their application without deploying it to the system. Flowdroid is one of the open-source static analysis tool for Android apps. Zhiyong Shan et, al., introduced KREfinder for static analysis to help find and reproduce incorrect handling of instance data[5]. With a test set of 324 apps, potential errors were found in 12 apps with a detailed overview on the error report. Despite being

used by the developers to find the errors in the app, static analysis can also help in evaluating the performance of the app. Though this is an effective way to test the mobile applications, there are decent amount of errors that cannot be detected during development phase. DroidDLP tool will help in finding such errors that has not been detected during the development, by using the UI testing in an automated approach.

CHAPTER 3

PROPOSED SYSTEM

After laying out the motivation for building a tool to test for the data persistency in the previous chapter, this chapter provides some details about the system architecture of the tool and its components. The proposed system contains a Test tool developed in Java, APKLists.xml file with the necessary data related to APKs, GooglePlayStore to download and save the required APKs. An overview of system architecture is shown in fig.

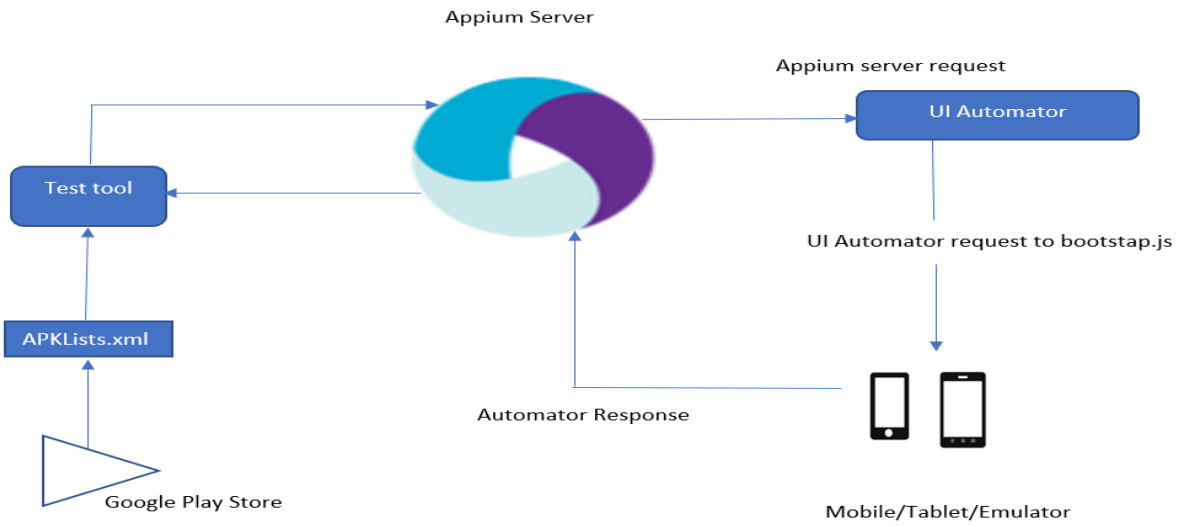


Figure 4: DroidDLP Design Overview

ApkLists.xml file is required to contain the list the apps and the respective apk details such as ActivitiyName (Main activity of the app which is the first activity of the application when it is launched), packageName (name of the package given by developer to the APK store) and the file location of the APK. Android Emulator or Android device connected to the computer and the ADB should be up and running. Once the setup is ready, Appium server is started on port 4723 and listens to the requests. Using Android UIAutomator in the Appium client, the tool is developed

using Java language. Several APIs of Appium are utilized while developing the tool, such as Android Mobile element driver to create a session between Appium and the ADB for communicating with the mobile device/emulator.

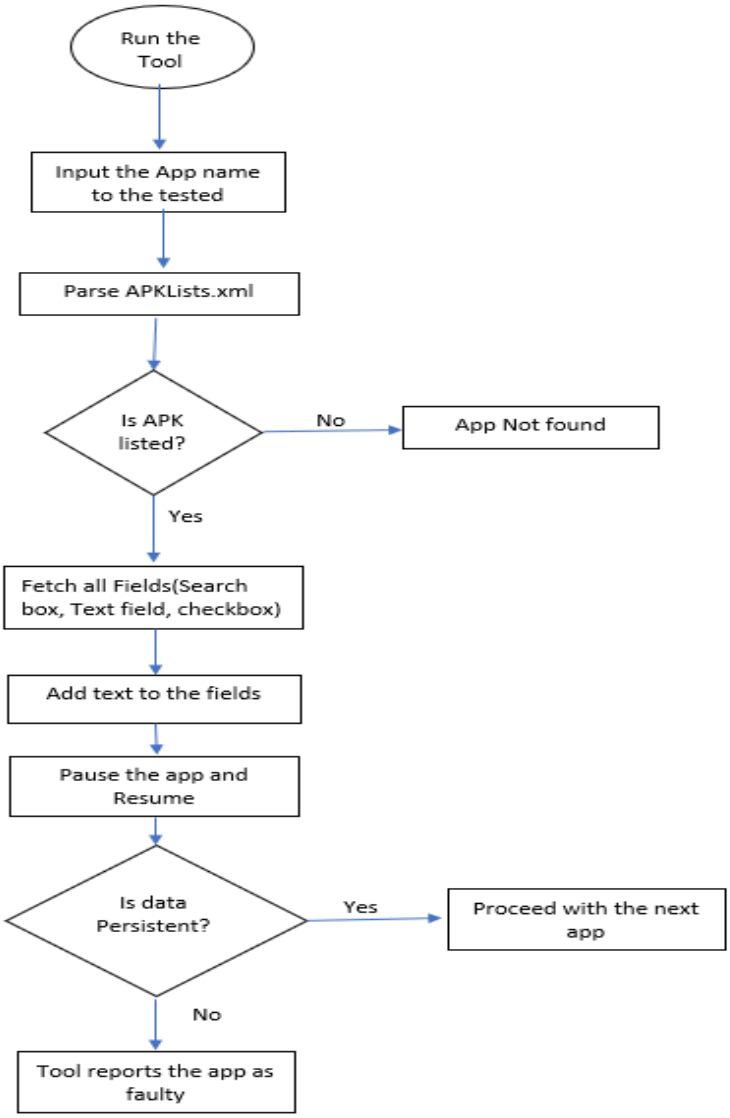


Figure 5: DroidDLP Flowchart

When DroidDLP tool is initiated, the first activity to input the app name that should be tested. Once the app name is provided, it will be passed to the XML parser to fetch the respective apk details. If the app is not found in the xml, then the tool with exit with error. User should enter

the app details to the xml file before testing it using DroidDLP. After fetching the APK details, with the use of Appium driver, the fields in each page will be gathered. DroidDLP tool will enter some test data to the fields that needs to be tested with some dummy values. Once the values are entered the onPause() -> onResume() state change is tested by sending the app to the background and bringing back to foreground. After that, the data will be validated with the data entered before and the data fetched after the resume. If there is data mismatch, then the tool will consider the app as faulty and displays the result.

CHAPTER 4

IMPLEMENTATION

Different packages were used to implement the DroidDLP tool. In this chapter, we will be discussing the implementation details and the source code developed to perform different tests in this tool from the core to final product in detail.

In general, GUI test automation is a complex task with some pre-conditions. Automation needs to exactly follow the same steps, like how humans interact with the GUI by using the buttons, widgets and by the flow of the app. There are several APIs provided by the applications such as UIAutomator, Expresso and so on to inject user inputs, such as pressing the keys or tapping the screen based on the coordinates provided by the API can often be easy, but interpreting information from complex graphical representations can sometimes be difficult [12], [13]. Our tool mainly uses UIAutomator for performing such activities to navigate or to give the input to the app.

In this research, the error types are categorized into 4 types each with a unique testing criterion and the apps are tested accordingly for recognizing these errors. Type 1 error deals with the text widgets, whenever text entered in the text widgets such as, signin page, Account details, messages and Registration windows is not persistent, it will be considered under this category. Likewise, when a selected checkbox gets unselected once the UI state transition is completed, it will be considered as Type 2. In the same way, when the search results of the text entered in the search bar clears after UI state transition, it is Type 3 error. Finally, when the activity of the mobile app before the UI state itself if different from the activity after the operation, that will be categorized as Type 4 error. The screenshots of the test results for these errors are explained in the later section.

DroidDLP tool is designed to take an application name as input and search for the app in the list of apps available already in the system with the corresponding APK details. To perform this task, whenever user runs the tool, it will prompt to input the app to be tested, XML parser of the tool will parse the APKLists.xml file that is present inside the project folder and gets the required details such as ActivityName, packageName and the file location of the APK which are required to install the APK on the Android device. The format of the xml file is specified in Fig: APKLists.xml file content format.

App	apkFile	appPackage	appActivity
ebay	ebay.apk	com.ebay.mobile	com.ebay.mobile.activities.MainActivity
lipsi	lipsi.apk	com.lipsisoftware.lipsi	com.lipsisoftware.lipsi.activity.LoginActivity
onelife	onelife.apk	com.motionvibe.onelife	com.mediapark.motionvibe.LoginActivity

Figure 6: APKLists.xml file content format

Appium requires Desired capabilities to be set to open a client connection to communicate with the mobile device. The information fetched from the first step using the XML parser will now be used to create a session with the mobile device that is registered for testing the app. The below code will help achieve this by setting the Desired Capabilities.

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability( MobileCapabilityType.AUTOMATION_NAME,
AutomationName.ANDROID_UIAUTOMATOR2);
capabilities.setCapability(MobileCapabilityType.DEVICE_NAME, “Android
Emulator”);
capabilities.setCapability(MobileCapabilityType.APP,
app.getAbsolutePath());
cap.setCapability(“appPackage”, packageName);
```

```
cap.setCapability(“appActivity”, activityName);
cap.setCapability(“automationName”, “uiautomator2”);
```

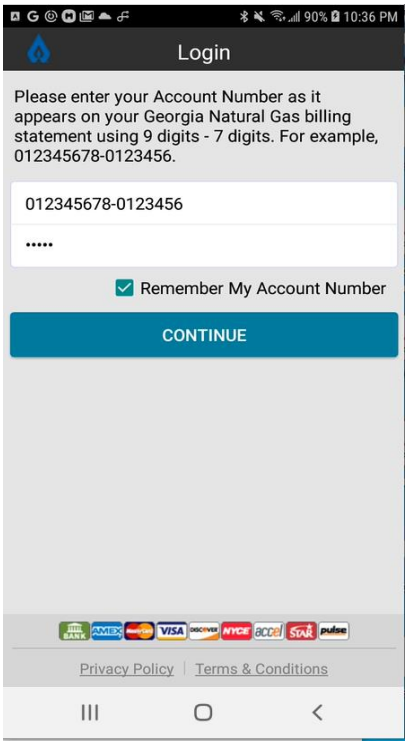
The app that is specified by the user will then be installed through Appium on the mobile device/emulator once the and the Android driver is created with all the required details as parameters. The android driver is responsible for running Appium settings on the mobile device, which will install and open the app and will check if the mainActivity is same as the activity that is the first screen of app. Each activity page will have some clickable widgets that are responsible for navigating through user’s command. In the tool, all these widgets were captured using “driver.findElementsByAndroidUIAutomator(“new UiSelector().clickable(true)”);”. Individual elements of a current page are stored in a list and will be passed to the function called parseActivities which iterates through all the elements and checks if there are any button widgets to click which navigates it to the next activity. Each activity page is monitored to fetch the Checkbox, Search Bars and Text Fields.

Test date will be entered in each of these fields and then the onPause()-> onResume() state change will be tested by clicking the home button and reopening the app in 5 seconds.

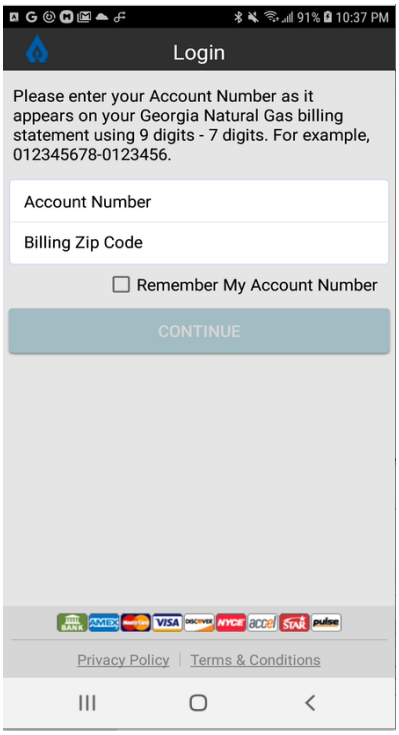
```
List<MobileElement>
tfl=driver.findElements(By.className(“android.widget.EditText”));
tfl.get(0).sendKeys(testString);
```

When the app is moved to the foreground, the data persistency is checked by fetching the text field value or the current checkbox selection of the respective fields using “driver.findElements(By.className(“android.widget.EditText”)).getText()”. The data retrieved is compared with the user entered data and if there is a mismatch, the testcase fails. In fig a and fig b, the Account number and billing zip details entered by user are not present once the app is moved

to pause state and resume. This also has one more scenario in which the checkbox selected to remember the account details also got unselected during the test.



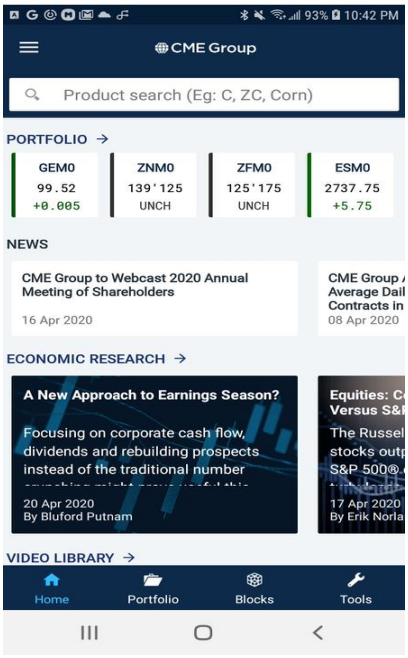
a) Error Type 1 and 2 (Before)



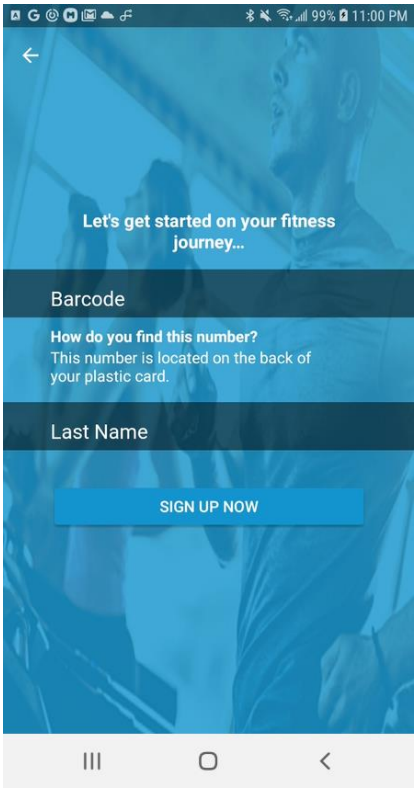
b) Error type 1 and 2 (After)

Likewise, fig c and fig d, also corresponds to a failure scenario, whereas this time, the failure is related to the search bar field. In this case, the search results of the stocks that are entered by the user in the search bar are lost once the onPause -> onResume state transition happens. Adding to this, fig e and fig f also have a issue, but in this case it is related to the transient UI state, once the app is sent to background, it will always reopen the app from the start which means that it is always launching the mainActivity of the app by deleting all the app data. This should be happening when the app is restarted or killed, but in this case, it is reflecting such scenario. This case is tested by comparing the activity before the state transition with the activity that is active after the transition. This is a generic use case that is tested before each type of error scenario, which voids the rest of the testing since it overwrites every value.

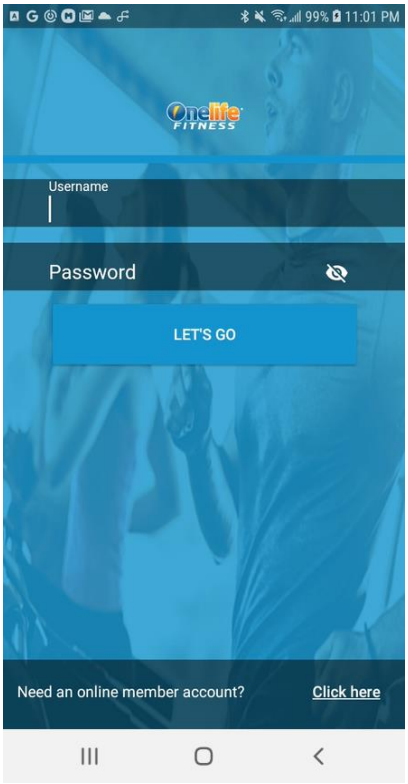
c) Error Type 3 (Before)



d) Error Type 3 (After)



e) Error Type 4 (Before)



f) Error Type 4(After)

Figure 7: Screenshots of error scenarios

The Junit framework integrated to the DroidDLP tool is responsible for comparing the values of the fields using assert statements and will give a detailed overview of the Test with the failure Trace.

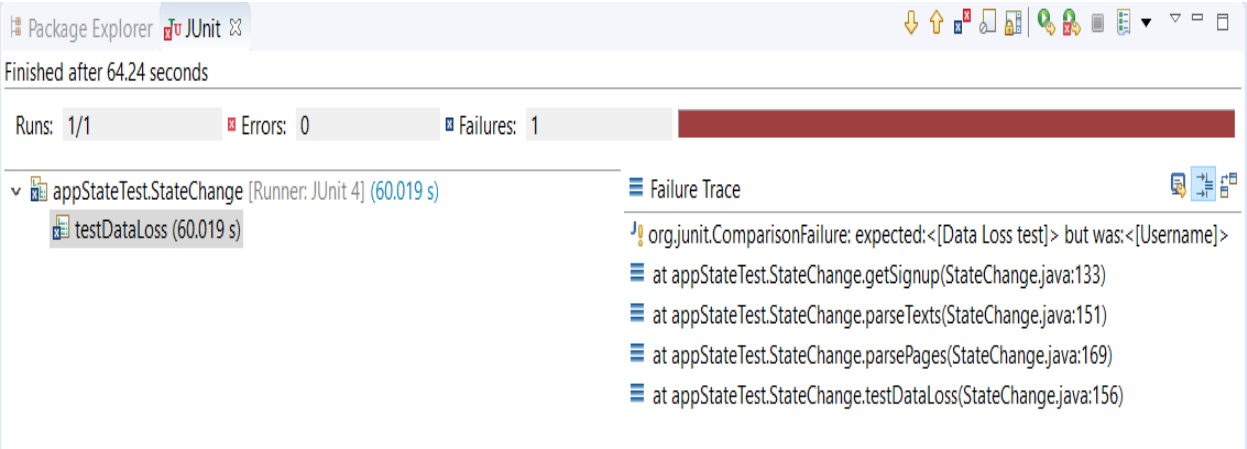


Figure 8: Failure Trace using JUnit

CHAPTER 5

EVALUATION AND RESULTS

The primary objective in evaluating the DroidDLP tool was to measure its efficacy in finding the data persistence in the apps.

5.1 Test Environment:

TABLE 1

EXPERIMENTAL ENVIRONMENT OF DESKTOP PC AND ANDROID DEVICE

Desktop	Operating System	Windows 10 Enterprise 64-bit
	Processor	Intel(R) Core(TM) i7 Desktop CPU @1.90GHz
	Memory	16 GB
Android Device	Android Virtual Device	Nexus 5X
		Android 8.1(API Level 27)
	Mobile Device	Samsung J7 Crown
		Android 8.0(API Level 25)

For evaluation, we used Samsung Galaxy J7 crown. We programmed all the modules in Java. The apps were installed on the on the smart phones with the latest Android Oreo 8.0 (API level 25) version. The tool is also tested on two virtual devices using Android Studio, with the Android Oreo 8.1(API Level 27) on Nexus5X and with Android Pie 9.0 (API level 28) version. We installed all the APIs and libraries in Android Studio and compiled using JRE 1.8.0. and JUnit 4.12.0. The Desktop Personal Computer used for testing this tool is having Windows 10 Enterprise

operating system. This is used to run the Appium server and Android Studio to connect the Mobile device through USB. Communication layer and every other module in our testing framework was implemented in Java.

5.2 Data Set:

Most of the apps in Google Play Store or APKPure are not available for the public for free and some other apps have different testing approach. For example, apps which stream online videos and movies require more memory and it is difficult to check the UI states in such apps. Some apps even prompt for verification to check if the app is getting accessed using an automated system or a human being, by giving object identification as prerequisite condition for login. Thus, all these types of issues are making it difficult for the tester to choose the apps without these kinds of errors or preconditions.

For testing our DroidDLP tool, we collected 395 apps without these issues and tested those apps for the error types that are mentioned in the previous chapter. Most of these apps are selected because of the generic implementation without object verification during registration or the app test. In addition to this, these apps can be accessed using the test accounts created. Once these accounts are created, these apps are tested using DroidDLP tool and we found 12 apps with the error cases. The average running time of the testcase to find the errors is 93s, calculated by comparing the test results of all the apps which were found using DroidDLP tool. Most of these apps have many installs which illustrates the applicability of our analysis on popular apps. Table 2 contains the list of apps which has the UI transition state errors, for each app, the name of the app, size (in MB) , number of installs as per Google play store and the time taken to run the testcase using DroidDLP.

TABLE 2

APP CHARACTERISTICS

AppName	Size	Installs	Time(in seconds)
FanLabel	170 MB	100000	95
CME Group Mobile	25 MB	50000	78
Connected2.me	80 MB	10000000	125
First Missouri Mobile	78 MB	100	65
FSCB Mobile Banking	17 MB	10000	78
GTE Cards	63 MB	5000	62
Bleacher Report	90 MB	1000000	210
MyOutdoorTV	81 MB	100000	95
SpendMgmt	94 MB	10000	170
OneLifeFitness	17 MB	10000	61
Lipsi	46 MB	100000	80
GNG Payments	3.9 MB	10000	75

5.3 Effectiveness and Performance Testing:

All the apps are tested manually by installing and running the app and check if there is any data loss during resume which will give the actual true positives. Following this approach is a standard to make sure that the false positives are the false negatives are categorized properly. The results of this calculations are listed in Table 3. Like many other tools, false positive cases are

observed during the test and the reason for that is recorded during our testing. Some of the applications are following the security protocol to delete the information provided by user once the app is sent to the background[14]. These apps include the banking app which need an Account number as the input to login. If the login is not processed, the details entered by the user are deleted once another activity is started or the current activity is sent to pause and resume state. Since this is a requirement for the app to implement security feature, this cannot be considered as a bug. In addition to this, when another app is run, there is a crash which resulted in reopening the app, but the tool assumed that the data is lost because of which there is a false positive case. The False positives and False negatives based on the error type are listed in Table 3 in addition to the precision and recall values. This gives a detailed overview of the performance of the tool.

TABLE 3

EFFECTIVENESS EVALUATION ANALYSIS

Error Type	# Apps with data Loss	False positives	False Negatives	Precision	Recall
Type 1	8	2	0	80%	100%
Type 2	3	0	0	100%	100%
Type 3	3	0	0	100%	100%
Type 4	2	1	0	66.6%	100%

TABLE 4

PERFORMANCE EVALUATION ANALYSIS

All units in seconds				
Error Type	Total Time	Average time per app	Maximum time per app	Minimum time per app
Type 1	876	95	150	62
Type 2	354	115	210	80
Type 3	340	112	170	74
Type 4	120	65	70	50

Out of 395 apps, 12 apps are reported as the apps with data loss and the efficiency data for those apps based on the time taken by error types are listed in Table 4. Out of all the listed error types, Type 3 error took more amount of time per app to find the error. This is because the app will take some time to fetch the search results before performing the actual test and also the navigation to the search bar will also consume some time. The Type 4 takes very less time to get observed because the error is generic and can be found from the second activity itself when the app is sent to background, and the activity will start from homepage. This does not require any specific wait time or search time. In addition to this data, Figure 9 compares the results of number of apps with a specific error type in a bar chart format.

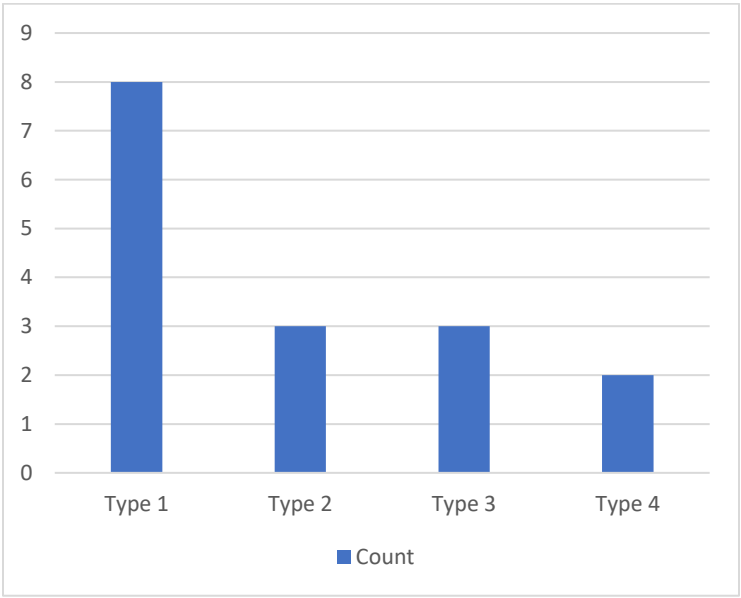


Figure 9: Apps count with respect to Error Type

CHAPTER 6

LIMITATIONS AND FUTURE WORK

Most of the applications on the internet will require authentication when we want to install and run the app. This limitation makes it difficult to login to the apps using test account to test the data. Though this is a limitation for the test user, a developer can always test their work using DroidDLP tool using the admin account. The testing of this tool can be extended to wide range of apps instead of limiting to 395 applications. The tool is now limited to search for only search boxes, signup and text fields and checkboxes and then apply the state transition test to find the rate of data persistency in the app. In future, this can be improved to search for other information such as testing the messengers, banking apps and other apps which deal with large amounts of data. This tool can be further developed to automatically searching for the APK details of the app from the internet instead of providing it by the user. The DroidDLP tool developed in this research can be possibly applied to different platforms and it will be able to provide more information if adapted to a web application.

CHAPTER 7

SUMMARY

This research has successfully developed a tool for automated dynamic detection of inadequate preservation of data during state change by installing apps that user chose to test in the android device/emulator. It has presented a motivations example where it is possible to see situations from real applications where the data is not persistent when its state changes. Based on the high F-measure rates and the time that it took to run these tools, we conclude that the tool is both effective and efficient. This tool can be added to the automated testing tools to test the apps irrespective to the requirements of the project and the functionalities provided by the app. Potentially, these kind of tools can be used to analyze an application prior to its publishing to the Android marketplace, saving time and money by avoiding having multiple version releases for the bug fixes.

REFERENCES

REFERENCES

[1] A. C. R. Paiva, J. M. E. P. Gouveia, J. Elizabeth and M. E. Delamaro, "Testing When Mobile Apps Go to Background and Come Back to Foreground," 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Xi'an, China, 2019, pp. 102-111.

[2] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito, "Software testing of mobile applications: challenges and future research directions", In Proceedings of the 7th International Workshop on Automation of Software Test (AST '12). IEEE Press, iscataway, NJ, USA, 29-35, 2012.

[3] "ViewModel Overview", https://developer.android.com/topic/libraries/architecture/viewmodel, accessed in 2020

[4] Android platform: "Saving UI State", https://developer.android.com/topic/libraries/architecture/saving-states, accessed in 2020.

[5] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding resume and restart errors in Android applications. SIGPLAN Not. 51, 10 (October 2016), 864–880. DOI:https://doi.org/10.1145/3022671.2984011

[6] Android platform: "Understand the Activity Lifecycle", https://developer.android.com/guide/components/activities/activity-lifecycle?hl=ru, accessed in 2020.

[7] Android developers homepage, https://developer.android.com/guide/, accessed in 2020

[8] D. Amalfitano, N. Amatuccia, A. Memon, P. Tramontana, A.R. Fasolino, "A general framework for comparing automatic testing techniques of Android mobile apps", in Journal of Systems and Software, Vol.125, pp. 322–343, March 2017.

[9] Mario Linares-Vsquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Crdenas, and Denys Poshyvanyk. "Enabling mutation testing for Android apps". In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). ACM, New York, NY, USA, 233-244, 2017.

REFERENCES (continued)

[10] Eugene Spafford, “Extending mutation testing to find environmental bugs”, Software: Practice and Experience, 20(2), 1990.

[11] Lin Deng and Jeff Offutt and Paul Ammann and Nariman Mirzaei, “Mutation operators for testing Android Apps”, in Information & Software Technology, vol.81, pp.154–168, 2017.

[12] C. Kaner, J. Bach, and B. Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach. Wiley, 2001.

[13] M. Fewster and D. Graham, Software Test Automation: Effective use of test execution tools. Addison–Wesley, 1999.

[14] L. Baird, Z. Shan and V. Namboodiri, "Automated Dynamic Detection of Self-Hiding Behavior," 2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems Workshops (MASSW), Monterey, CA, USA, 2019, pp. 87-91.