An Investigation into the Effectiveness of Automated Testing Methods for Android Applications Using FATTDroid Framework

Oluwasogo Adekunle Okunade, Ph.D.[1*]; Christiana Uchenna Ezeanya, Ph.D.[1]; Emmanuel Gbenga Dada, Ph.D.[2]; Adako Kwanashie, M.Sc.[1]; Oluwatoyosi Victoria Oyewande, M.Sc.[1]; and Oke Eke Ndukwe M.Sc.[1]

[1]Department of Computer Science, Faculty of Computing, National Open University of Nigeria, Abuja, Nigeria.
[2]Department of Computer Science, Faculty of Physical Sciences, University of Maiduguri, Maiduguri, Nigeria.

E-mail: aokunade@noun.edu.ng *

ABSTRACT

In the dynamic landscape of mobile application development, automated testing methods play a pivotal role in ensuring software quality and reliability. Because the mobile industry is so competitive, developers of mobile applications now pay special attention to the quality of the apps they create so that it will promptly respond to any negative reviews, and regularly update the products to address any bugs that users may have found. As a result, testing these apps has become essential and is thought to be the greatest way to guarantee that software is of a high standard in terms of features, functionality, behaviors, and performance.

This paper presents an in-depth assessment of the efficiency of automated testing methods for Android applications. Through a comprehensive evaluation process, including the execution of various testing techniques on selected Android applications and analysis of test results, the study aims to provide insights into the strengths, weaknesses, and overall effectiveness of these methods using FATTDroid framework. Key metrics such as fault detection rates, false positive rates, and resource consumption are examined to gauge the performance of each testing approach.

Our findings show no significant difference exists between the systematic techniques adopting either the Breadth First or Depth First exploration techniques. We also observed that techniques adopting the MAV method are more accurate and cover more lines of code than their SAV-based counterpart. However, they are more expensive than the technique with fewer lines of code. Regarding the choice between the systematic and random techniques, our finding indicates that the random techniques are more effective, covering more lines of codes than the systematic techniques. However, the data indicates that they are always more expensive. The findings of this research contribute to a deeper understanding of automated testing methodologies in the context of Android app development, offering valuable guidance for developers and testing practitioners seeking to optimize their testing strategies and enhance software quality.

(Keywords: Android mobility, automated testing, smartphones)

INTRODUCTION

In the year 2021, about 1.54 billion smartphones were sold to users worldwide and over 371 million smartphones were purchased in the fourth quarter of 2021 (Counterpoint, 2022; Statista, 2022b). The high demand for smartphones and tablets led to a corresponding increase in the development of mobile applications (web and native apps) for such devices to deal with the computational needs of their users (Gao, *et al*., 2014; Salihu, *et al*., 2019).

A recent study shows that over 230 billion mobile applications (Mobile Apps) were downloaded worldwide in 2021 alone (Statista, 2022a). The rate at which these mobile applications are being published into the marketplace poses a great challenge for software engineering researchers (particularly software testers) in determining the quality of these apps (that is, their functionality, behaviors, and performance under certain

conditions) (Salihu, *et al*., 2019; Yang, *et al*., 2015). Some of the challenges that hinder or limit the abilities of the software testers to effectively verify the quality of application software are the fragmentation of device models, variety of operating system (OS) platforms, fast release cycles, a huge number of mobile network operator, time and language barriers (such as: time zone variation and targeted users), data security breaches, etc. (Gao, *et al*., 2019; Ma, *et al*., 2016; Sauce Labs, 2017).

Mobile applications (otherwise known as Mobile Apps) are software programs that run on mobile devices such as smartphones, tablet PCs, and other mobile devices with the capability to run network-based applications over a cellular or satellite data link (Eke and Salihu, 2021; Myers, *et al*., 2012). Due to the competitive nature of the mobile industry, mobile application developers now pay close attention to the quality of applications they develop, response time to any form of negative reviews, and constantly updating their apps in other to rectify any bug-related issues reported by the users of these applications.

Therefore, testing of these applications has become necessary, and is considered the best mechanism to ensure the quality of software programs in functionality, behaviors, performance, and features (Tao and Gao, 2016). According to Morgado, *et al*. (2014), testing is essential in improving the quality of a product, hence, a crucial part of the mobile development process. The goal of testing software applications is to discover any form of defect that may affect the general functionality of the system. Testing can be *manual* – requiring some level of human intervention to generate test cases or interpret test results or *automatic* – automatically generate and execute test cases.

Automated testing has emerged as an indispensable practice in the development lifecycle of Android applications, offering developers a systematic and efficient way to verify the functionality, performance, and reliability of their software. Automated testing plays a crucial role in ensuring the quality and robustness of Android applications. By automating repetitive testing tasks, developers can detect bugs and issues early in the development process, leading to faster time-to-market and improved user satisfaction. Moreover, automated testing enables continuous integration and delivery practices, facilitating rapid iteration and deployment of app updates.

Researchers and practitioners have explored various methods, approaches and techniques for testing mobile apps recently which has resulted in the development of several testing tools. These tools implemented different testing strategies that evaluate apps at various levels of testing. However, each tool has its strengths and weaknesses and is suitable for testing different quality factors. Some of these testing techniques rely largely on manual efforts during setup, others are fully automated, but report limited bugs due to their inability to test some unique features of an app, while some require some kind of effort in interpreting the test result (Amalfitano, *et al*., 2017a). This has led to difficulties in determining which testing techniques should be considered for testing mobile apps, therefore, there is a need for empirical criteria to classify testing techniques, to guide practitioners in selecting suitable ones. This overview provides insights into the significance of automated testing in Android app development, its key components, benefits, and challenges. The study will analyze criteria for classifying mobile apps testing tools/techniques the use FATTDroid framework to assess and compare testing techniques for Android mobile apps based on the criteria commonalities

LITERATURE REVIEW

The authors of this paper offer a review of the current research literature related testing techniques in the field.

Testing Techniques

Testing of mobile applications in order to assess or verify their quality can be done either manually or automatically. Researchers have explored and documented various kinds of testing techniques in the cause of identifying and implementing a test strategy that is more efficient and cost-effective. Generally, test cases are important software testing inputs that must be generated from some information that is some type of software artefact. Various types of artefacts are utilized as reference inputs for generating test cases. These artifacts encompass the program source code, software specifications and/or design models; and information dynamically

obtained from program execution. The various types of testing techniques are – *Script-based, Capture/Replay, Random walk, Model-based, Search-based, Symbolic/Concolic Execution, and Combinatorial-based* (Anand, *et al*., 2013; Salihu, *et al*., 2017). These techniques have further been discussed in the following sub-sections and the studies that explore them identified accordingly.

Script-Based Technique

The script-based technique requires the software tester to manually write the test scripts that exploit the features of the application under test (AUT). A recent study conducted by Pan, *et al.* (2022) presents *METER (Mobile Test Repair),* a script-based technique and tool for repairing GUI test scripts for mobile apps that leverages computer vision techniques. Imparato (2015) proposed a combined GUI Ripping technique with input perturbation testing, a technique that systematically explores the behavior of Android applications and creates a model of the explored GUI and then uses it to generate the perturbed test inputs; this technique was implemented in a tool called *GUI Ripper*.

Jiang, *et al*. (2007) presented a script-based technique implemented in the *MobileTest* tool that adopts a sensitive-event-based approach to simplify the design of cases and enhance the efficiency and reusability of the case.

Capture and Replay

The Capture/Replay technique records entries during a manual test and creates automated test scripts that can be reused. This technique is considered the early transition stage from manual to automated testing, it records sequence of user interactions with an application and then converts them into test scripts that can be automatically replayed. Capture/Replay can otherwise refer to as C*apture and Playback or Record and Replay*.

Gomez and Millstein (2013) presented *RERAN,* a capture-and-replay technique that directly captures and replays the low-level events like touch screen gestures - *tap, swap, pinch* and *zoom* that are triggered on the device and other sensor input devices such as light sensor, compass, and accelerometer.

Liu, *et al*. (2014) proposed a methodology for capturing user interactions and generating test scripts based on the replay phase that supports assertions that aid in detecting errors during test case execution, the approach was implemented in a tool called *Android Capture and Replay Test* tool (ACRT).

Lin and Chiao (2014) present *Smart Phone Automated GUI* (*SPAG*), a capture and replay tool that is based on an image processing technique and dynamically changes the timing between events to adapt to the workload of the device, their technique is implemented in a tool known as *SPAG-C*, an improved version of the *SPAG* tool. Compared to the SPAG, *SPAG-C* reduces the time of the testing process and increases usability (Lin and Chiao, 2014).

Random Walk

In the random testing technique, the system under test (SUT) is tested by generating random and independent inputs and test cases. This type of testing technique like the one proposed by Hu (2011) are mostly implemented in *Monkey Tool,* although they are considered very scalable and easy to use, its effectiveness can be questionable (Patel, *et al*., 2018) because Monkey Tool can be modified to allow certain adjustment to different classes of events (i.e., a tester can decide to send "*click"* events on the GUI more often than the *gesture* events) (Amalfitano, *et al.*, 2017).

Liu, *et al*. proposed *Adaptive Random Testing (ART),* a random testing technique for testing mobile applications as well as interactive systems like TV and PlayStation. This technique uses two testing metrics: the *F-measure – the number of test cases required to detect the first failure* and *the time used to find the first fault.* It sends sequences of random events that are farthest away from the already executed testing sequences following a specified distance metric for event-drive systems (Liu, *et al.*, 2010). The sequences of events are randomly generated until a predefined length is reached.

MacHiry, *et al*. (2013) presented *Dynodroid* tool, a technique for generating inputs for mobile app testing is based on an *"observe-select-execute"* circle. Dynodroid first *observes* events that are relevant to the app in the current state, then randomly *selects* one of the events, and *executes*

the selected event to produce a new state (Machiry, *et al.*, 2013). This test execution circle stops after executing a predefined number of events.

Morgado, *et al.* (2016) employ a random technique that randomly fires events to a running mobile application; their technique which was implemented in the *iMPAcT tool* dynamically analyses the AUT, identifying specific UI Patterns of the app and checking whether they are properly implemented. The testing process is terminated after the execution of a predefined number event. Another random technique relying on the Monkey Tool was proposed by Zhauniarovich, *et al.* (2015) and implemented in the *BBoxTester* framework.

Model-Based

In the model-based testing technique, test cases are derived from a model that describes the functional aspects of the application under test. All model-based techniques are automated; they include any approach that automatically generates and executes test cases without any human intervention. Amalfitano, *et al.* (2014) presented *MobiGUITAR* (enhanced version of Android Ripper) an automated GUI-driven testing tool for Android apps that is based on the observation, extraction, and abstraction of the run-time state of GUI widgets.

Salihu, *et al.* (2019) proposed *AMOGA*, a tool for automated UI model generation for mobile applications. *AMOGA* employs a static-dynamic approach, the static approach extracts the mobile application events statically by analyzing the byte code, while the dynamic approach matches the event list of UI elements associated with the event dynamically to explore the application's behavior.

Azim and Neamitu (2013) present a testing strategy that employs two systematic testing techniques (*Targeted Exploration and Depth-First Exploration)* and is presented in a tool known as the *A^3E tool. Targeted Exploration technique* - (a direct approach) initially, uses the static bytecode analysis to extract a *Static Activity Transition Graph (SATG)* and then systematically explores the running app. The *Depth-First Exploration technique* employs an automated depth-first approach to explore the Activities and GUI elements of the AUT. The *Dynamic Activities Transition Graph (DATG)* extracted from the technique is made up of a set of vertices/nodes and edges (Vs, Es), where the vertices or nodes "Vs" represents the app activities, while the set of edges "Es" is the actual activity transition observed at runtime.

In another study, Arnatovich, *et al.* (2018) present *mobolic* an automated GUI testing tool for mobile applications. *Mobolic* uses a combination of online testing and customized input generation techniques to automatically generate test cases with high coverage.

Another technique implemented in the *AndroidRipper* tool in (Amalfitano, *et al.,* 2011, 2012, 2014, 2015 and 2016) has been attributed to Amalfitano, *et al.*, these techniques automatically and systematically traverse the apps, firing a predefined number of events to the application under test to restrict and maintain a GUI Tree model of the graphical user interface, then the navigate through the model until all distinct GUIs of the AUT are reached.

Maiya, *et al.* (2014), in their study "Race Detection for Android Applications" presented a technique implemented in a tool called *DROIDRACER* that systematically tests mobile apps and detects data races by computing the 'happens-before' relation on the traces. DROIDRACER analyzes the AUT at runtime and systematically extracts UI event sequence in a depth-first approach. This technique does not infer a model of the Application under test, it stops after a predefined length of execution.

Wen, *et al.* (2015) proposed a technique that dynamically analyzes the AUT using the master-slave approach. Their technique is implemented in a tool called *PATS (Parallel Android Testing System)* (Wen et al., 2015). PATS uses a GUITree for implementing a breadth-first traversal strategy, firing events such as *button clicks* and *screen-scrolling* on the GUI. The test execution terminates once there are no more user interface *(UI)* states of the GUITree to visit.

Active Learning-Based

Model learning (Choi, *et al.*, 2013) addresses the shortcomings of model-based testing by learning a model of the GUI app during a test to guide the generation of user input sequences based on the model. Choi, *et al.*, introduced an active learning technique implemented in a tool called *SwiftHand* (Choi, *et al.*, 2013). This technique utilizes

execution traces generated during the testing phase to learn an approximate model of the graphical user interface (GUI), known as the *Extended Deterministic Labeled Transition System (ELTS).* The acquired model is subsequently utilized to select user inputs that navigate the application to states that have not been explored previously. SwiftHand implements two traversal strategies based on the L* algorithm for the learned model.

Search-Based

The search-based testing technique involves the use of a metaheuristic *(a higher-level procedure designed to find, generate, or select a heuristic that may provide a sufficient solution to a given problem)* search optimizing technique (like Genetic Algorithm) to fully or partially automate the generation of test data (Mcminn, 2004; Mcminn, *et al.*, 2004).

Mahmood, *et al.* (2014) designed and presented the *EvoDroid tool,* a technique that employs an automated technique to extract two models, namely *the Interface Model (IM)* and *the Call Graph Model (CGM),* from the source code of the application under test (AUT). The IM represents the external interfaces of the application, while the CGM captures its internal behaviors. By utilizing the CGM, *EvoDroid* performs a segmented search, identifying the code segments that can be explored independently. It evaluates the quality of different test cases by assessing the paths they traverse through the CGM (Mahmood, *et al.*, 2014).

Yang, *et al.* (2013) proposed a search-base technique implemented in the *ORBIT tool* that automatically obtain the GUI model of a mobile app by analyzing the app's source code statically, enabling the extraction of set of user actions supported by each widget within the GUI then, systematically exercise these events on the running app using the depth-first search strategy.

Wang, *et al.* (2014) present a technique in a tool known as *DroidCrawler,* that automatically explores the AUT using a depth-first search strategy to extract the GUI Tree model from the graphical user interface.

Symbolic/Concolic Execution

In the *symbolic* testing technique, test input values are represented with *symbolic values* instead of concrete (actual) data and program variables as *symbolic expressions* (Păsăreanu and Visser, 2009), hence, producing output values as a function of the input symbolic values. Whereas *concolic execution* automates test input generation by the concrete (actual data) and symbolic values for the inputs and executes the program both concretely and symbolically (Sen, 2007).

Anand, *et al.* (2012) proposed an automated approach that is based on concolic testing, their testing technique is implemented in a tool called *ACTEve.*

Gao, *et al.* (2018) present a concolic execution technique for Android apps that automatically deduces expression representing Android models dynamically at the time of execution, the technique is implemented in a tool known as *SynthesiZE*. The authors noted that the major problem with concolic execution is the risk of getting stuck while navigating many program paths before reaching the targeted state. On the contrary, *SynthesiZE* takes a more practical approach by concretely executing a program and progressively refining the synthesized function. This iterative process continues until the generated inputs successfully reach the targeted states (Gao, *et al.*, 2018).

Combinatorial Based

Combinatorial testing otherwise known as *Combinatorial Interaction Testing,* tests the system under test (SUT) with all the required parameter value combinations; this type of testing technique can detect failure triggered because of the interactions among parameters in SUT (Nie and Leung, 2011; Shiba, 2004).

Wang, *et al*. (2020) proposed *ComboDroid,* a combinatorial-based testing technique that obtains use cases for manifesting a specific app functionality (either manually or automatically extracted), and systematically identifies the combinations of use cases, thereby, providing high-quality test inputs.

Huynh, *et al.* (2019) presented a combinatorial testing technique for test cases and test data generation for mobile applications. This technique has been implemented in a tool called *CTGen; CTGen* utilizes the *In-Order-Parameter-General (IPOG) Algorithm* that can edit the model files, configure the input parameters, and generate unit tests (Huynh, *et al.*, 2019).

Technique Categorization Based on Degree of Automation

This section categorizes the testing techniques presented above based on their degree of automation (i.e., test execution and test case generation). Some techniques use only the test execution step while requiring some level of human intervention at the test case generation step. We refer to this set of techniques as *partial or semi-automated techniques* (examples include the *Script-based* and *Capture/Replay* techniques).

Others are *fully automated techniques* that provide a greater degree of automation since they can automatically generate and execute test cases concurrently. The essential difference among these sets of fully automated techniques lies in the time spent on generating test cases and executing them. Test case execution according to Utting, *et al.* (2012) can either be done offline or online. In an offline scenario, test cases are generated before they being executed, they generally exploit the existing model of the AUT and can use any of the techniques as identified in section 2.4 to generate text cases. While in online technique both the test case generation and execution are done iteratively in runtime.

Techniques within this category according to (Amalfitano, *et al.*, 2017) can be classified into two groups mainly, techniques implementing *Random* selection of events and techniques that follow a more *systematic* approach. The systematic approach begins by launching the execution of the application under test (AUT), and then iteratively fires events to it.

Comparing Different Testing Techniques

Due to the increased number of proposed testing techniques addressing the automated testing of mobile applications, researchers have begun to evaluate these testing techniques and the tools that implement them to ascertain which technique is most preferable for testing mobile apps. Some of these researchers based their comparison on tools that test different techniques, while others compare different techniques implemented by the same tool.

For comparisons among tools, Choudhary, *et al.* (2016) conducted a comparative study of the existing test input generation tools for Android apps, the testing tools used in this study implemented different testing techniques, they include random (Monkey, Dynodroid, PUMA), model based (GUIRipper, ORBIT, SwiftHand, A^3E), and concolic testing (ACTEve). The authors' comparison metrics are based on ease of use, ability to work on multiple platforms, code coverage, and fault detection. The experiment was conducted on 68 real-world mobile applications using an emulator, configured to allow each tool to run for 1 hour on each app.

Yang, *et al*. (2013) compared the performance of four different tools (*Orbit, Monkey, AndroidGUITAR, and Android GUI Ripper*) implementing different testing techniques (search-based, random, model-based, and script-based), respectively.

Others focus on the testing techniques rather than the tools implementing them, these include *PUMA* (Hao, *et al.*, n.d.), a programmable UI automation framework used for implementing several monkey testing techniques. PUMA allows the tester to specify navigation hints for app exploration and the logic for analyzing the app properties.

Machiry, *et al.* (2013) compared the code coverage and bug detection capabilities of their proposed tool Dynodroid against the Monkey tool, using both the *Frequency, Uniform, and Biased* random testing approach.

Choi, *et al.* (2013) in their comparative study believe that their active learning technique is the application GUI achieves more code coverage than the random and L* techniques. Among all the studies, none undergo any form of systematic empirical validation in obtaining their result; therefore, the result obtained could be influenced by the selected termination strategy since the execution of the test case is stopped after a predefined number of events is reached.

METHODOLOGY

We identify and extract common features otherwise referred to as parameters in our work and they include *Exploration Method, Learning Method, Extraction Method, Scheduling Method* and *Termination Method.*

Exploration Method*:* this parameter is used to define the approach used by the fully automated technique while exploring the AUT and for generating test cases. The potential values for this parameter include – *random* and *systematic* respectively. The random technique does not rely on the AUT model while the other techniques exploit the model of the AUT at runtime.

Systematic Method*:* this parameter is valid if the Exploration Method requires or relies on the model of the AUT. It is made up of two sub-parameters – *Inferred Model* and *Abstraction Method*. The former states the type of GUI model inferred by the testing technique.

The five main inferred models identified in the literature are the *GUITree, DATG, SATG* and *EDLTS.* While the latter, describes the method employed to create an abstract representation of a GUI model, they include the Active Name (AN), a GUI state that belongs to the same "*Activity class*", the "*Single Attribute Value*" (SAV) refers to a scenario in which a GUI state within the model is linked to GUIs that share an identical set of widgets and possess the same value for a particular attribute while the "*Multiple Attribute Values"* (MAV) refers to a situation where a GUI state is associated with all the GUIs that have the same set of widgets and exhibit identical values for a subset of the widgets' attributes.

SAV and MAV can be configured to allow the software tester the flexibility to select the specific type of widget and attribute to consider while describing the GUI.

Termination Criterion: this parameter defines the approach used to terminate or end an automated testing process. The common Termination Criterion values that can be assumed by any online testing technique include a Predefined Number of Events (PNE) that stops the iteration loop after executing a predefined number of events, a Predefined Length (PL) that stops the loop when the sequence of events generated reached a predefined length, a Predefined Time of Execution (PTE) stops the loop once a predefined time of execution is reached. However, in a situation where a random exploration method is used, the 'Sat' value terminates the testing process when N (a given random technique) reaches the same testing adequacy i.e., the same code coverage. On the other hand, the MC value terminates the process when an expected coverage has been reached.

Extraction Criterion: this parameter defines the event(s) to be extracted and sent to the AUT by the technique. We identified two values for this purpose: the Predefined Events (PE) and the Relevant Events (RE) values, the former considers a subset of the predefined type of event that could be potentially managed by any Android application, while the latter ensures that the technique extracts only relevant events that can be handle by the app in its present state.

Scheduling Method: this parameter explains the method used to select the sequence event(s) to be fired next during the test process. The two possible sets of values as identified in the literature are *graph-based* and *random-based values,* the former is common with the fully automated techniques. They include *L*, Targeted, Depth First, Breadth First* and *L* - minimal restart.* The latter involves random selection of events such as *Priority, Frequency, Adaptive, Biased* and *Uniform.*

Uniform Representation of Techniques

In this section, we represent the online testing techniques using a uniform notation referred to in this study as Testing Technique Representation (TTR) which comprises the instances of the following 6-ple:

TTR = (Exploration Method, Inferred Model, Abstarction Method, Termination Criterion, Extraction Criterion, and *Scheduling Method)*

where:

Exploration Method ε {Random, Model-based, Learning-based, Search-based, Concolic/Symbolic Execution, and Combinatorial};

Inferred Model ε {FSM, EDLTS, SATG, DATG, fGFG and GUITree};

Abstarction Method ε {MAV, SAV, AN};

Termination Criterion ε {MC, PTE, PL, PNE, Sat};

ExtractionCriterion ε {PE, RE};

Scheduling Method ε {L*, Targeted, Depth First, Breadth First, and L* - minimal, Biased-random, Frequency-random, Priority-random, Uniform-random and Adaptive-random};

With this representation, all testing techniques having the above parameter values can be unambiguously described based on our TTR characterization. This will help easy classification and identification of the commonalities and differences that exist among the testing techniques.

Modeling Effectiveness and Cost

We analyze the effectiveness of the online testing technique using the code coverage metrics. The choice for choosing this indicator is due to its popularity and it's considered one of the most frequently used methods for evaluating the testing process. The percentages of the source code executable statements covered by the technique at the termination point were considered. Then, we give the following definitions:

· *T:* is the current technique.

· *AUT*: is the application under test

· *#LOCs (AUT)*: the number of executable Lines of Code consisting of the source code of the *AUT.*

· *#CovLOCs(T, AUT):* is the number of LOCs belonging to the *AUT* and covered by the technique *T.* A line of code is covered if it has been covered at least once by *T.*

· *Cov(T, AUT):* is the percentage of the *AUT* statements Covered by the technique *T* at its termination point.

Due to our inability to identify a suitable measure that can effectively demonstrate the desired outcome of implementing a particular technique until it reaches completion.

We decided to evaluate the cost of technique *T* for testing an AUT by considering the total number of iterations *I* required for the algorithm's loop to reach its termination point.

This serves as an indicator able to express the required effect for executing a technique till it reaches the termination point. Expressed as follows:

Cost(T, AUT) = I

We intentionally did not regard the execution time as a cost indicator due to potential bias by the developmental (e.g., design decisions) and technological (e.g., testing infrastructure and machine) choices made for implementing and executing a given technique. The two dependent variables *Cov(T, AUT)* and *Cost(T, AUT)* as identified in this section will be used to answer our research questions.

Experimental Setup

We selected a sample of 15 open-source applications from the Google Play Store for this experiment, downloaded the apk file version of the app and extract the *source code* using a combination of tools (such as dex2jar and JD-GUI tool). Here, only the number of *Classes, Methods* and *Lines of Code (LOC)* were considered for our experiment.

Table 1 shows a list of the sample applications with their corresponding information.

Experimental Procedure

The experimental procedure involves three steps. First, we set up a testing environment and then implement the five-technique representations TTR9, TTR10, TTR11, TTR12 and TTR6 for this experiment. Table 2 presents the testing techniques involved.

Then we implemented the techniques in our FATTDroid framework ensuring they all follow the same strategies such as (i) the insertion of input values in text fields in the GUIs and (ii) the timing between events to be fired. We injected random values that were visited most during the navigation into text fields. Then set 2 seconds constant time delay between consecutive events. Afterwards, the Emma library tool was used to measure the code coverage achieved by a testing technique, the bytecode of the Android application under test.

Table 1: Sample Applications and Characteristics.

ID	Application	Description	#Classes	#Methods	#LOC
A1	Tomdroid 0.7.1	Note taking	133	707	3860
A2	BatteryCircle 1.8.1	Battery state viewer	13	71	249
A3	Simply Do 0.9.2	Task list manager	46	246	1281
A4	Trolly 1.4	Shopping list manager	19	64	364
A5	Pedometer 1.4.0	Steps counter	28	223	809
A6	Man Pages 1.51	Unix man pages viewer	11	49	292
A7	Bites 1.3	Recipes manager	40	139	967
A8	WorldClock 0.6	Regional clock	18	83	1149
A9	MunchLife 1.4.4	Game points counter	10	28	184
A10	QuickSettings 1.9.9.3	System settings manager	90	514	2811
A11	Aarddict 1.4.1	Dictionary and Wikipedia reader	93	424	2097
A12	TicTacToe 1.0	Simple game	13	47	493
A13	JustSit 0.3.3	Chronometer	11	61	273
A14	TaskMan 1.01	System task manager	3	24	226
A15	TinyTipper 1.2	Tip calculator	133	707	3860

Table 2: Testing Techniques Involved in the Experiment.

Technique	Exploration Method	Inferred Model	Abstraction Method	Termination Criterion	Extraction Criterion	Scheduling Method	Testing Technique Representation ID
T1	Systematic	GUITree	SAV	MC	RE	BF	TTR9
T2	Systematic	GUITree	MAV	MC	RE	DF	TTR10
T3	Systematic	GUITree	SAV	MC	RE	BF	TTR11
T4	Systematic	GUITree	MAV	MC	RE	DF	TTR12
T5	Systematic	None	None	Sat	RE	Uniform	TTR6

Test Technique Execution

We tested the 15 sample applications on an emulated Android device by launching the applications on the Android Virtual Device (AVD). The AVD was configured to emulate a Galaxy Nexus device with 512 MB of RAM and 2GB memory size with the Marshmallow operating system. Then we allowed the testing technique to the execute to termination point. When execution did not terminate within 2 hours, we proceed to analyze the partial results (i.e., achieved code coverage, the inferred GUI model, and the characteristics of the explored GUIs) to check if the GUI model being inferred was exploding. If so, then we forced the termination of the testing execution.

Data Collection

After the test reaches its termination point, we collect the code coverage reports generated by the Emma tool and evaluated the dependent variable *Coverage.* To measure the *Cost* variable, we extract the number of iterations performed by the Algorithm. Furthermore, we filter out the data associated with the testing executions that were forcefully terminated.

Result Analysis

Here, we present and analyze the data obtained for each of our dependent variables.

Table 3: Coverage and Cost Results per Technique.

	T1		T2		T3		T4		T5	
	Cov	Cost	*Cov*	Cost	*Cov*	Cost	*Cov*	Cost	*Cov*	Cost
A1	35.63	137	44.29	471	34.91	134	38.00	475	69.96	147800
A2	91.24	26	91.24	26	91.24	26	91.24	26	92.89	500
A3	78.54	187	*	*	76.30	187	*	*	85.10	4700
A4	61.35	58	73.71	77	60.60	58	74.26	243	80.88	600
A5	65.54	48	66.49	82	65.54	48	66.74	82	74.75	4,600
A6	76.15	70	76.16	84	76.16	70	76.16	73	76.85	700
A7	59.41	229	61.28	879	58.89	209	61.8	775	62.5	16,700
A8	86.54	13	89.18	147	86.54	13	89.18	147	97.37	3,600
A9	78.09	29	93.42	8,902	78.09	29	93.42	8,902	98.86	700
A10	39.67	254	43.61	591	39.67	254	43.61	590	48.5	30,700
A11	44.25	49	44.25	49	44.25	49	44.25	49	71.14	4,000
A12	78.5	69	82.35	130	78.48	69	81.74	127	99.64	20,100
A13	62.05	51	67.61	330	62.05	48	67.61	593	70.92	5,300
A14	91.01	28	91.01	50	91.01	22	91.01	43	92.79	800
A15	75.34	81	*	*	75.34	81	*	*	87.86	7,200

Table 3 presents the *Cov* and *Cost* values of the five techniques derived from the sample applications. Here, the AUT is represented by the *A1 - A15* in the vertical axis and the code coverage achieved in the horizontal bars.

A graphical representation of this report is shown in Figure 1. Here, the AUT is represented by the *A1 - A15* in the vertical axis and the code coverage achieved in the horizontal bars. The horizontal bars are represented in different colours, each bar representing a different technique. While Figure 2 shows the testing *cost* of the different testing techniques. As shown in the Figure, the random technique is the most expensive.

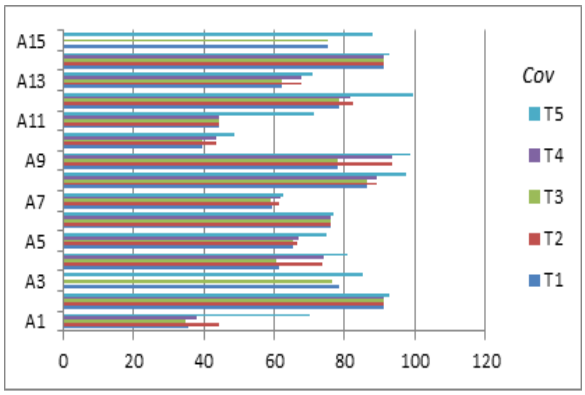


Figure 1: Coverage (Cov) of the Techniques T1, T2, T3, T4 and T5.

Analyzing Code Coverage

We compare the coverage of two techniques by evaluating the difference in the code coverage percentages of the two techniques (*Ti* and *Tj*) used for testing the same *AUT* using the formula:

$\Delta cov(Ti, Tj, AUT) = Cov(Ti, AUT) - Cov(Tj, AUT)$

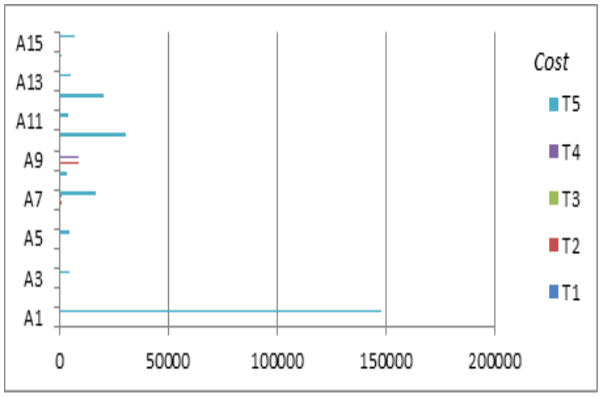
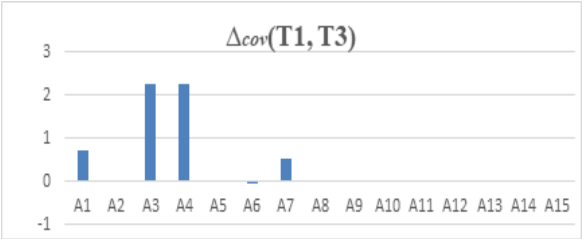


Figure 2: Cost of the Techniques T1, T2, T3, T4 and T5

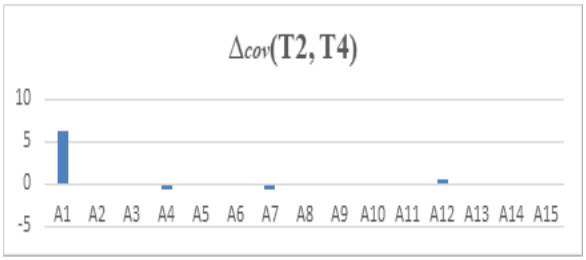
To answer the research question (*RQ1.1),* we compared the performance of systematic techniques that exploit the *BF* scheduling method against the ones that use the *DF* value. We considered two separate comparisons in order to answer this question. First, we consider techniques exploiting the *SAV* value in the *Abstraction Method* parameter and measure the difference in the code coverage percentage *(T1, T3, AUT).*

Figure 3(a) presents the results of this comparison. Finally, we considered techniques exploring the *MAV* value, note the difference code coverage for *(T2, T4, AUT)* and then report our findings in Figure 3(b).

Figures 3(a) and 3(b) represent differences between the code coverage of *SAV* and *MAV* values. The above graphs which have on the horizontal axis, the sample applications used and on the vertical axis the differences between the code coverage values of the two techniques have shown that most of the applications have no significant difference in code coverage between the *BF* and *DF* testing techniques despite the abstraction method they adopt. In Figure 3(a), 11 applications out of the 15 under test report equal code coverage values while in Figure 3(b), 10 out of the 15 apps reported equal code coverage.



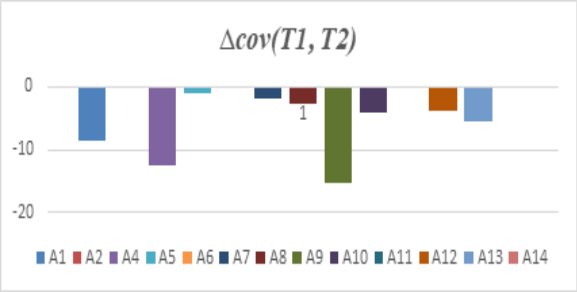
3(a) Techniques adopting the SAV value of the Abstraction Method parameter



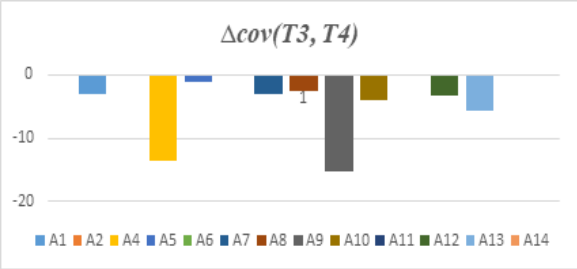
3(b) Techniques adopting the MAV value of the Abstraction Method parameter

Figure 3: Difference in Code Coverage between BF and DF Techniques

To answer *RQ2.1,* we evaluate the performance of the systematic techniques adopting the *SAV* values as against the systematic techniques using the *MAV* values. Here, we also considered two comparisons. First, techniques adopting the same *BF* value of the exploration method parameter and measuring the $\Delta cov(T1, T2, AUT)$, report our finding in Figure 4(a), then we took a similar approach for techniques with *DF* values, measuring the $\Delta cov(T2, T4, AUT)$ and show our findings in Figure 4(b). Here, we observed a different pattern taken by the graph. This is because the *MAV-based* technique (*T2* and *T4*) covered more codes than the *SAV-based* ones (in 9 apps over 13 both with *BF* and *DF*) with a difference of 5.7% on average. There is no difference in coverage between *MAV-based* and *SAV-based* techniques in the remaining 4 applications. Therefore, the data shows that the techniques adopting the *MAV-based* abstraction method are more effective than the ones adopting the *SAV-based* abstraction method.



4(a) Techniques adopting the *BF* value of the *Exploration Method* parameter



4(b) Techniques adopting the *DF* value of the *Exploration Method* parameter

Figure 4: Difference in Code Coverage between SAV-Based and MAV-Based Techniques.

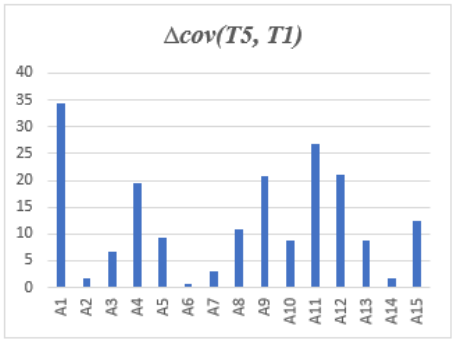
To address research question *RQ3.1* as to how effective a technique adopting the *Random* exploration method is compared to the technique adopting the *Systematic* exploration method. We compare the code coverage reached by the *random* technique *(T5)* against the one achieved by the *systematic* techniques *(T1, T2, T3* and *T4)* and report our findings in Figure 5 (a – d).

According to the graph, the difference in code coverage between the techniques is always greater than zero, with a minimum value of 0.7% for application A6 and a maximum value of 35% for application A1. This data shows that randomly sending events to the AUT proved to be more efficient than employing a systematic approach for event generation where both techniques are executed until the termination point.

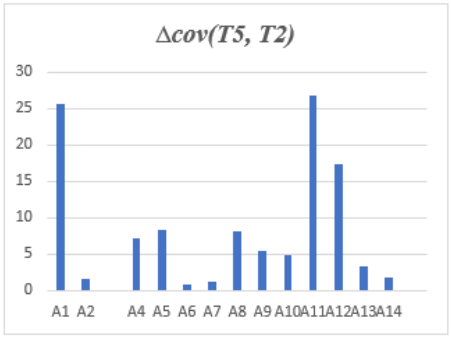
Analyzing the Cost

We compare and evaluate the cost required by two techniques *Ti* and *Tj* for testing the same AUT using the *Δcost(Ti, Tj, AUT)* metrics:

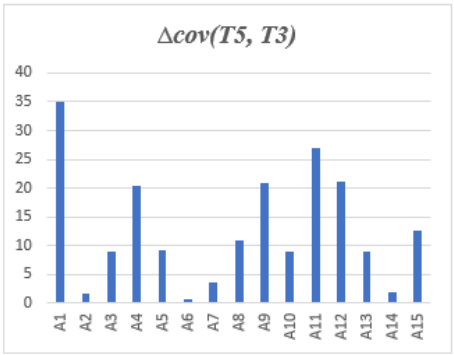
Δcost(Ti, Tj, AUT) = Cost(Ti, AUT) – Cost(Tj, AUT)



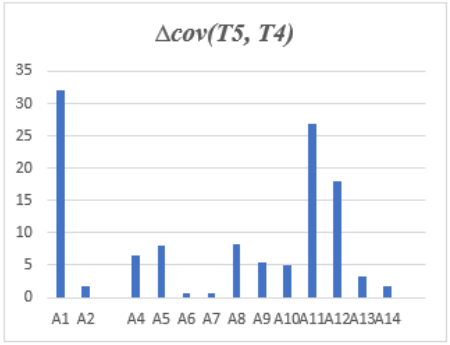
(a) Difference in code coverage between T5 and T1



(b) Difference in code coverage between T5 and T2



(c) Difference in code coverage between T5 and T3



(d) Difference in code coverage between T5 and T4

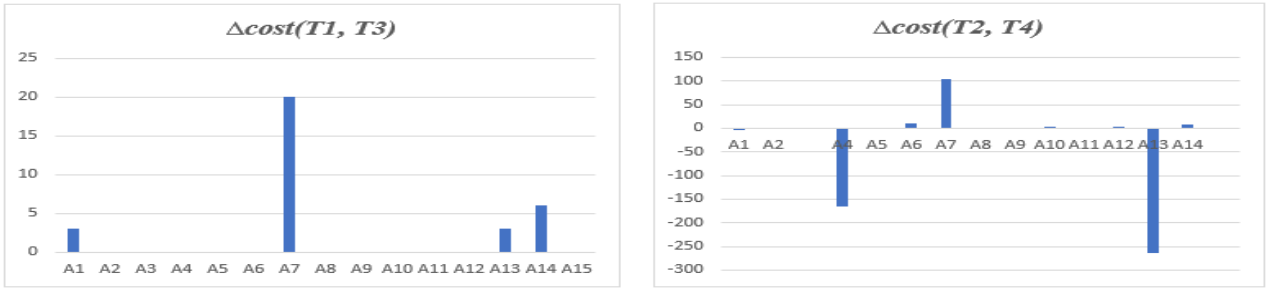
Figure 5: Differences in Code Coverage between Random and Systematic Techniques.

In other to address *RQ1.2,* we compare the *Δcost(T1, T3, AUT)* techniques adopting the *SAV* value of the abstraction method parameter with that of the *Δcost(T2, T4, AUT)* using *MAV* value and report our findings in Figure 6.

As observed in Figure 6(a), most of the applications show no difference in the cost among *BF-based* and *DF-based* techniques using the *SAV* abstraction method. *T1* and *T3* had the same cost in 11 applications out of 15, while in the remaining 4 applications *(A1, A7, A13, A14), T1* proves to be more expensive than *T3.* The average difference in cost between the techniques is 2.1 iterations.

Figure 6(b), indicates that *BF* and *DF-based* techniques require almost the same cost, 9 applications out of 13 reported zero or less than 10 iterations. *T4* was more expensive than *T2* in apps *A4* and *A13,* whereas *T2* was reported as more expensive than *T4* in apps *A6* and *A7* with an average difference of 24 iterations. This indicates that there was no difference in the cost between *BF-based* and *DF-based* techniques despite the abstraction method they assume.

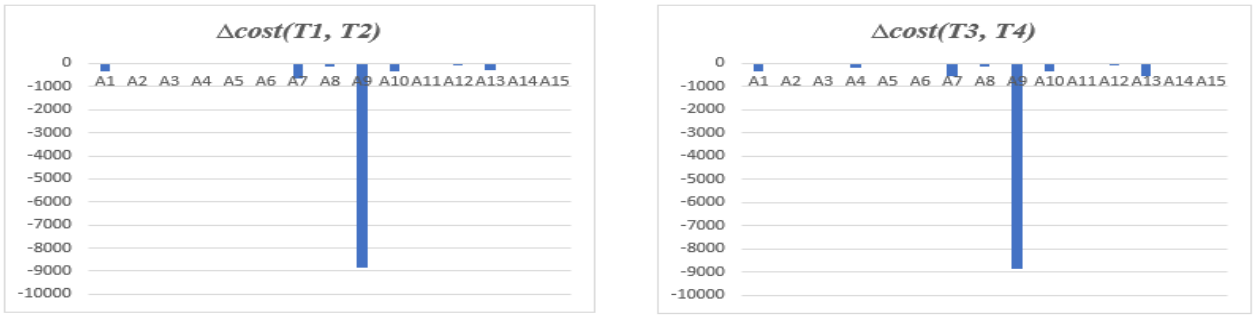
For *RQ2.2,* we compared the costs between *BF* and *DF* techniques when they assume both the *SAV* and *MAV* abstraction values respectively. Figures 6(a) and 6(b) report the difference. The charts in Figures 7(a) and 7(b) indicate that the *MAV* techniques are more expensive compared to the *SAV* ones in 22 cases out of 26. We observed that *A2* and *A11* of both the *SAV* and the *MAV* techniques have equal costs. This data shows that techniques adopting the *MAV* abstraction method were always more expensive than their *SAV* counterpart.



(a) Techniques adopting the *SAV* abstraction method parameter

(b) Techniques adopting the *MAV* abstraction method parameter

Figure 6: Difference in Cost between BF-Based and DF-Based Techniques.



(a) Techniques adopting the *BF* Exploration Method parameter

(b) Techniques adopting the *DF* Exploration Method parameter

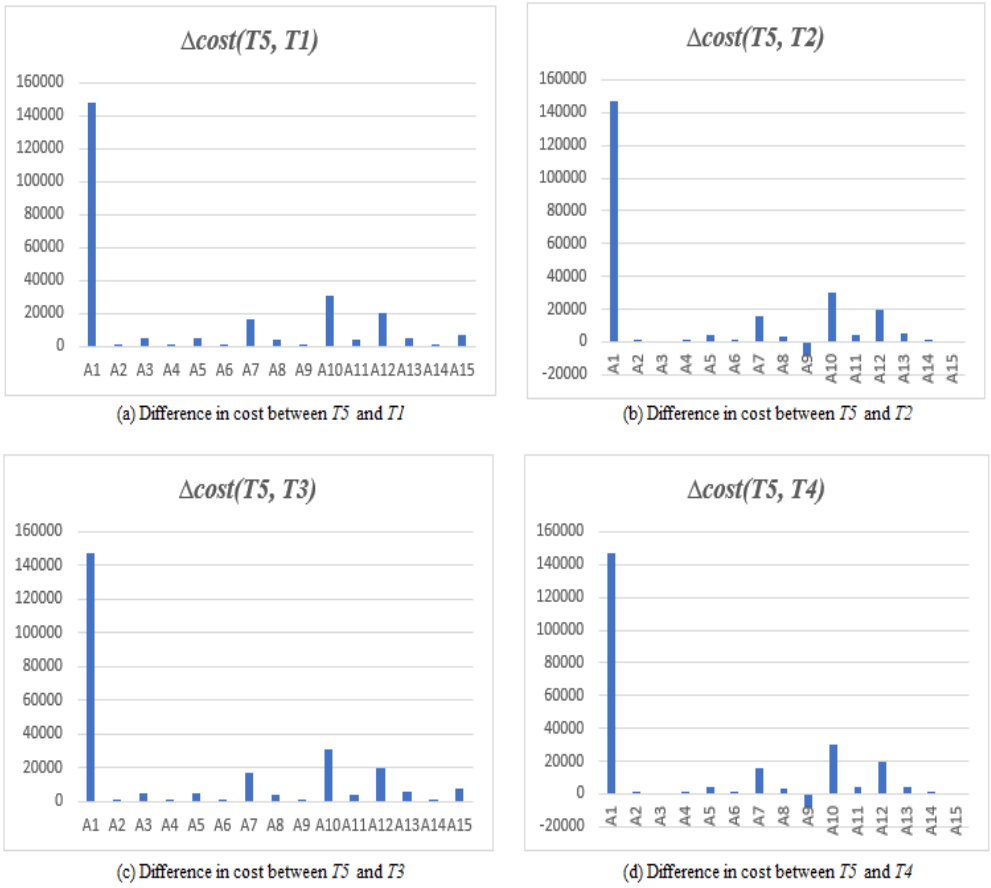
Figure 7: Difference in Cost between SAV and MAV-Based Techniques.

To answer question *RQ3.2,* we conduct four comparisons to ascertain the cost of the *random* technique *T5* as against the *systematic* techniques *(T1, T2, T3* and *T4),* then, report the difference in cost among the techniques in Figures 8(a – d). Our findings based on the graph indicate that the random technique is more expensive compared to all the systematic techniques, with an average cost of more than 16,700 iterations. Only in a single case with *A9* application in *MAV-based T2* and *T4* cost approximately 13 times more than the random *T5* technique. This data shows that the *random* technique is more expensive than the systematic technique for generating events when they both have to execute to the termination point.

DISCUSSION AND FINDINGS

The fully automated testing techniques for Android applications considered in this research are a representation of popular techniques proposed in the literature and also implemented in a tool. Our major concern is to understand the effects different implementations of these techniques might have on their performance and cost. Our findings show no significant difference exists between the systematic techniques adopting either the *Breadth First* or *Depth First* exploration techniques. Based on this, we conclude that there is no need to consider any of the exploration methods over the other.

We further investigate the techniques exploiting different methods of analyzing the state of the GUI. We compared the *SAV* (a method considering a single attribute of the GUI widget) over the *MAV* (a method that considered multiple attributes of the GUI widget). We observed that techniques adopting the *MAV* method are more accurate and covers more line of code than their *SAV-based* counterpart. However, they are more expensive than the technique with fewer lines of code.



(a) Difference in cost between *T5* and *T1*

(b) Difference in cost between *T5* and *T2*

(c) Difference in cost between *T5* and *T3*

(d) Difference in cost between *T5* and *T4*

Figure 8: Differences in Cost between Random and Systematic Techniques.

Regarding the choice between the *systematic* and *random* techniques, our finding indicates that the *random* techniques are more effective i.e., covering more lines of codes than the systematic techniques. However, the data indicates that they are always more expensive.

Threat to Validity

The limitations of our empirical study are as follows:

i) Construct validity threats – this relates to the appropriateness of our measures for capturing the independent variables. In this study, we used code coverage to measure the effectiveness of the testing techniques; other indicators such as the fault-detection ability of the techniques could also be considered. We adopted an approximation of the real cost of executing a testing technique metric for evaluating the cost of a technique. The number of restarts of the AUT or the number of events generated by the algorithm and fired to the AUT could be considered more accurate measures.

ii) Internal validity threats – this relates to conditions that are likely to affect the dependent variables of the experiment. We chose a single state of each app and set it before executing each considered technique without considering other states of the applications. This may have an impact on the code coverage and cost we reported.

iii) External validity threats – this threat questions the ability to generalize the results of our findings. Due to the difficulties in obtaining the application's source code from the Google play store, we've applied the testing techniques to 15 open-source Android applications. The executable (.apk) files of some of these applications are gotten from websites such as *apkpure.com* and then re-engineered to access the source code. The choice of these applications does not in any way reflect the wide variety of available applications in the app market today. We cannot conclude that the techniques considered in this study will behave differently with other applications.

CONCLUSION

In this study, *FATTDroid,* framework was used for comparing different online testing techniques. *FATTDroid* utilizes a unified representation of techniques identifying the commonalities and differences of the existing techniques. Using this framework, we successfully re-cast the characteristics of the online techniques suggested in existing literature, and then select a subset of five techniques for our experiment. The experiment was conducted on 15 Android applications and the results shows that some aspects of the techniques should be considered while designing a testing technique as they have the potential to influence the efficiency and efficacy of the testing process while some factors do not necessarily affect the result of the test. The *FATTDroid* framework will enable mobile app developers or testers to systematically explore the parameter space of a given technique for the purpose of selecting a suitable technique for testing mobile apps.

REFERENCES

1. Amalfitano, D., N. Amatucci, A.R. Fasolino, and P. Tramontana. 2016. "A Conceptual Framework for the Comparison of Fully Automated GUI Testing Techniques". *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ASEW 2015, 50–57. https://doi.org/10.1109/ASEW.2015.19

2. Amalfitano, D., N. Amatucci, A.M. Memon, P. Tramontana, and A.R. Fasolino. 2017. "A General Framework for Comparing Automatic Testing Techniques of Android Mobile Apps". *Journal of Systems and Software*. 125: 322–343. https://doi.org/10.1016/j.jss.2016.12.017.

3. Ardito, L., R. Coppola, S. Leonardi, M. Morisio, and U. Buy. 2020. "Automated Test Selection for Android Apps based on APK and Activity Classification". *IEEE Access*. 8: 187648-187670.

4. Choi, W., G. Necula, and K. Sen. 2013. "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning". *ACM SIGPLAN Notices*, 48(10): 623–639. https://doi.org/10.1145/2544173.2509552

5. Choudhary, S.R., A. Gorla, and A. Orso. 2016. "Automated Test Input Generation for Android: Are We There Yet?". *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2015, 429–440. https://doi.org/10.1109/ASE.2015.89.

6. Counterpoint. 2022. “Global Smartphone Market Share: By Quarter”. 2022.

7. Eke, N.O. and I.A. Salihu. 2021. “Design and Implementation of a Mobile Library Management System for Improving Service Delivery”. *Path of Science*. 7(4): 3001. https://doi.org/10.22178/pos.69-7

8. Gao, J., X. Bai, W.T. Tsai, and T. Uehara. 2014. “Mobile Application Testing: A Tutorial”. *Computer*. 47(2): 46–55. https://doi.org/10.1109/MC.2013.445

9. Gao, R., Y. Wang, Y. Feng, Z. Chen, and W.E. Wong. 2019. “Successes, Challenges, and Rethinking – An Industrial Investigation on Crowdsourced Mobile Application Testing”. *Empirical Software Engineering*, 24(2): 537–561. https://doi.org/10.1007/s10664-018-9618-5

10. Hu, C. 2011. “Automating GUI Testing for Android Applications”. *Proceedings of the 6th International Workshop on Automation of Software Test*, AST’11, ACM, New York, NY, USA, Section 4, 77–83. https://doi.org/10.1145/1982595.1982612

11. Hu, G., X. Yuan, Y. Tang, and J. Yang. 2014. “Efficiently, Effectively Detecting Mobile App Bugs with appDoctor”. *Proceedings of the 9th European Conference on Computer Systems, EuroSys* 2014. https://doi.org/10.1145/2592798.2592813

12. Imparato, G. 2015. “A Combined Technique of GUI Ripping and Input Perturbation Testing for Android Apps”. *Proceedings - International Conference on Software Engineering*. 2: 760–762. https://doi.org/10.1109/ICSE.2015.241

13. Kong, P., L. Li, J. Gao, K. Liu, T.F. Bissyandé, and J. Klein. 2018. “Automated Testing of Android Apps: A Systematic Literature Review”. *IEEE Transactions on Reliability*. 68(1): 45-66.

14. Lin, Y. and N, Chiao. 2014. “Improving the Accuracy of Automated GUI Testing for Embedded Systems”. February, 39–45.

15. Liu, C., C. Lu, S. Cheng, K. Chang, and Y. Hsiao. 2014. “Capture-Replay Testing for Android Applications”. https://doi.org/10.1109/IS3C.2014.293

16. Liu, Z., X. Gao, and X. Long. 2010. “Adaptive Random Testing of Mobile Application”. *ICCET 2010 - 2010 International Conference on Computer Engineering and Technology, Proceedings*. 2: 297–301. https://doi.org/10.1109/ICCET.2010.5485442

17. Lv, Z., C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang. 2022. “Fastbot2: Reusable automated model-based GUI Testing for Android Enhanced by Reinforcement Learning”. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. pp. 1-5.

18. Machiry, A., R. Tahiliani, and M. Naik. 2013. “Dynodroid: An Input Generation System for Android Apps”. *2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2013 - Proceedings, 224–234. https://doi.org/10.1145/2491411.2491450

19. Maiya, P., A. Kanade, and R. Majumdar. 2001. “Race Detection for Android Applications”. ACM SIGPLAN Notices, 49(6), 316–325. https://doi.org/10.1145/2666356.2594311

20. Morgado, I.C. and A.C.R. Paiva. 2016. “The iMPAcT tool: Testing UI Patterns on Mobile Applications”. *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2015, 876–881. https://doi.org/10.1109/ASE.2015.96

21. Morgado, I.C., A.C.R. Paiva, and J.P. Faria. 2014. “Automated Pattern-Based Testing of Mobile Applications”. *Proceedings - 2014 9th International Conference on the Quality of Information and Communications Technology*, QUATIC 2014, 294–299. https://doi.org/10.1109/QUATIC.2014.47

22. Myers, G.J., T. Badgett, C. Sandler, G. Myers, , T. Badgett, and C. Sandler. 2012. “Mobile Application Testing”. *Art of Software Testing, 3RD Edition*.

23. Pan, M., T. Xu, Y. Pei, Y., Li, T. Zhang, and X. Li. 2022. “GUI-Guided Test Script Repair for Mobile Apps”. *IEEE Transactions on Software Engineering*. 48(3): 910–929. https://doi.org/10.1109/TSE.2020.3007664

24. Patel, P., G. Srinivasan, S. Rahaman, and I. Neamtiu. 2018. “On the Effectiveness of Random Testing for Android: or How I Learned to Stop Worrying and Love the Monkey”. *Proceedings - International Conference on Software Engineering*, 34–37. https://doi.org/10.1145/3194733.3194742

25. Pecorelli, F., G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba. 2022. “Software Testing and Android Applications: A Large-Scale Empirical Study”. *Empirical Software Engineering*. 27(2): 31.

26. Pecorelli, F., G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba. 2020. “Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps”. *Proceedings of the 28th International Conference on Program Comprehension*. 296-307.

27. Ran, D., Z. Li, C. Liu, W. Wang, W. Meng, X. Wu, ... and T. Xie. 2022, “Automated Visual Testing for Mobile Apps in an Industrial Setting”. *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. pp. 55-64.

28. Salihu, I.A., R. Ibrahim, B.S. Ahmed, K.Z. Zamli, and A. Usman. 2019. “AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing”. *IEEE Access*. 7: 17158–17173. https://doi.org/10.1109/ACCESS.2019.2895504

29. Salihu, I.A., R. Ibrahim, and A. Mustapha. 2017. “A Hybrid Approach for Reverse Engineering GUI Model from Android Apps for Automated Testing”. 9(3): 45–49.

30. SauceLabs. 2017. “Mobile App Testing: Main Challenges, Different Approaches, One Solution.

31. Statista. 2022. “Annual Number of Global Mobile App Downloads 2016 – 2021”. 271644.

32. Statista. 2022b. “Cell Phone Sales Worldwide 2007–2021”. 263437.

33. Sutter, T., T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein. 202. “Dynamic Security Analysis on Android: A Systematic Literature Review”. *IEEE Access*.

34. Wen, H., Y. Li, G. Liu, S. Zhao, T. Yu, T. Li, and Y. Liu. 2024. “AutoDroid: LLM-Powered Task Automation in Android”.

35. Yang, W., M.R. Prasad, and T. Xie. 2013. “A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications”. *Bibliometrics Data Bibliometrics*, 1–2.

36. Zhauniarovich, Y., A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci. 2015. “Towards Black Box Testing of Android Apps”. *Proceedings - 10th International Conference on Availability, Reliability and Security*, ARES 2015, 501–510. https://doi.org/10.1109/ARES.2015.70

AUTHOR CONTRIBUTION

Oluwasogo A. Okunade prepared the literature review and oversaw the article writing; **Christiana Uchenna Ezeanya** wrote the research methodology; **Emmanuel Gbenga Dada** performed fieldwork; **Adako Kwanashie** conducted the statistical analysis; **Oluwatoyosi Victoria Oyewande** interpreted the results; and **Oke E. Ndukwe**, who drafted the paper.

CONFLICTS OF INTEREST

The authors have no conflicts of interest to declare.

SUGGESTED CITATION

Okunade, O.A., C.U. Ezeanya, E.G. Dada, A. Kwanashie, O.V. Oyewande, and O.E. Ndukwe. 2024. “An Investigation into the Effectiveness of Automated Testing Methods for Android Applications Using FATTDroid Framework”. *Pacific Journal of Science and Technology*. 25(2): 64-80.