International Journal on Advanced Science Engineering Information Technology
Vol.10 (2020) No. 1
ISSN: 2088-5334

TEGDroid: Test Case Generation Approach for Android Apps Considering Context and GUI Events

Asmau Usman[#+], Noraini Ibrahim[+], Ibrahim A. Salihu[+&]

[#]*Faculty of Sciences, Department of Computer Science, Abdu Gusau Polytechnic Talata Mafara, 892, Zamfara State, Nigeria.*
E-mail: asmee08@gmail.com

[+]*Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia, Johor, Malaysia*
E-mail: norain@uthm.edu.my

[&]*Department of Software Engineering, Faculty of Natural and Applied Sciences, Nile University of Nigeria, Abuja, Nigeria*
E-mail: ibrahim.salihu@nileuniversity.edu.ng

***Abstract*— The advancement in mobile technologies has led to the production of mobile devices (e.g. smartphone) with rich innovative features. This has enabled the development of mobile applications that offer users an advanced and extremely localized context-aware content. The recent dependence of people on mobile applications for various computational needs poses a significant concern on the quality of mobile applications. In order to build a high quality and more reliable applications, there is a need for effective testing techniques to test the applications. Most existing testing technique focuses on GUI events only without sufficient support for context events. This makes it difficult to identify other defects in the changes that can be inclined by context in which an application runs. This paper presents an approach named TEGDroid for generating test case for Android Apps considering both context and GUI Events. The GUI and context events are identified through the static analysis of bytecode, and the analysis of app's permission from the XML file. An experiment was performed on real world mobile apps to evaluate TEGDroid. Our experimental results show that TEGDroid is effective in identifying context events and had 65%-91% coverage across the eight selected applications. To evaluate the fault detection capability of this approach, mutation testing was performed by introducing mutants to the applications. Results from the mutation analysis shows that 100% of the mutants were killed. This indicates that TEGDroid have the capability to detect faults in mobile apps.**

***Keywords*— context event; GUI event; mobile application test case generation; software testing.**

I. INTRODUCTION

In recent year's mobile apps are developed to address more critical areas of people's daily computing needs, which brought concern on the quality of applications. In order to build high quality and more reliable applications that can gain recognition in the high-level competitive application's (app) market, there is a need for effective testing techniques to validate the quality of the applications. The techniques should be able to validate different types of events supported by mobile apps [1-4] in order to improve users' confidence in the mobile apps [5-7]. Testing event-driven applications present a great challenge to software testers such as the need to generate a huge number of possible event sequences that could sufficiently cover the application's state space [8, 9].

Several techniques are used to generate test cases that can be run to detect faults. Test cases must be generated from some information, specifically some software artifacts. The software artifacts that is used includes: software specification documents/or design models; software program/source; information about input/output space, and information dynamically obtain from program execution [10]. Most of the approaches dedicated to mobile apps dynamically analyze an application to generate events sequence that can later be used as test cases to test an application.

Nowadays test automation is becoming increasingly popular among the software engineering community in recent times [11-13]. Numerous testing techniques have been proposed for testing mobile apps in the past few years. However, most of the testing techniques for mobile apps generate test cases considering only GUI events such as [14, 15] without sufficient support for testing context events [16, 17]. Therefore, it will be difficult to identify other defects in the changes in the contexts, which can be inclined by the context in which an application runs [16]. In order to ensure that these applications behave correctly, external context events must be considered during testing such as those from GPS location data, sensors, network in addition to the GUI events.

There are few testing approaches and techniques that addressed testing context events for mobile apps such as [16, 18-20].

A. Related Works

There are only few studies that considered context events in the literature. Smart-monkey [7] is a testing approach for mobile apps that integrates features of event-based testing with random testing. It uses an ART algorithm adapted from [21] to automatically generate test cases that are composed of both GUI events and context events based on event sequence distance it measure the distance between the test cases of mobile applications. The aim of the approach is to reduce both the number of test cases and the time needed to expose the first fault. Extended Ripper [18] is proposed for automating testing of Android apps which considers both context events and GUI events for testing Android mobile apps. It is based on reusable event patterns that were manually defined after an initial analysis conducted on the bug reports of open source applications. Test cases can be generated based on the defined event patterns using three scenario-based mobile testing approaches: manual, mutation-based and exploration based. Since the event patterns are derived manually by an expert from analyses of bug history, the events that may trigger a faulty behavior in an app may not be identified accurately. Moreover, when testing other types of applications, these event patterns may need to be redefined.

Greibe et al, [22] propose a model-based approach to improve the testing of context-aware mobile apps by deducing test cases from design-time system models. Android tool [23] implements a technique to address the context changes in mobile apps called block-based context-sensitive testing. Test cases are split blocks that can be reused and combined with context changes that are used in testing different scenarios without duplicating the test cases. This reduces the effort of writing cases. Yu and Takada [24] proposed an approach for generating test cases for both GUI and context events from android mobile apps using events pattern that are derived manually similar to that in [18], according to the authors their results were inconclusive because no bug was found for the set of generated test cases.

An approach is proposed by [16] to systematically generate several executing contexts from the permission of an Android app. Their approach analyses the lists of permissions and the resources that an application uses. Various contexts are generated from a mobile app by permuting resource conditions. The permutations of the contexts are prioritized and selected for test case generation. Few selected apps' permissions were considered to identify their related resources and possible states. As such some faulty behavior may not be detected and there is no clear means of detecting context from the dynamically changing environment.

Dynodroid [20] uses the Adaptive Random technique (ART) to generate a sequence of events that can be used to systematically explore an application. The approach is based on the observe-select-execute sequence to generate both user and system events by checking the ones that are relevant to the app. However, one of the limitations of the approach is the restriction of the apps under test from communicating with other apps. As many Android apps communicate with other apps for shared functionality and some context could not be detected.

EHBDroid [19] is an approach for testing Android apps that simulate a large number of events by invoking the event handlers directly. The event handlers of an activity are invoked randomly which only consider event coverage. The approach is not based on event sequence.

II. Material and Method

This paper focused on test case generation for mobile apps. Most mobile testing technique focused on GUI events only. This contributes to insufficient coverage of mobile apps during testing. While, testing context events of mobile apps have numerous challenges. The major challenge is how to identify the context events from an application. Thus, in this research both GUI events and context events supported by mobile apps are considered. The bytecode and manifest.xml file is analysed to extract GUI events and context events respectively. Considering GUI analysis in combination with mobile apps' permission analysis can generate comprehensive events that can be used to generate test cases in order to provide better test coverage. The TEGDroid consists of five main steps as shown in Figure1.

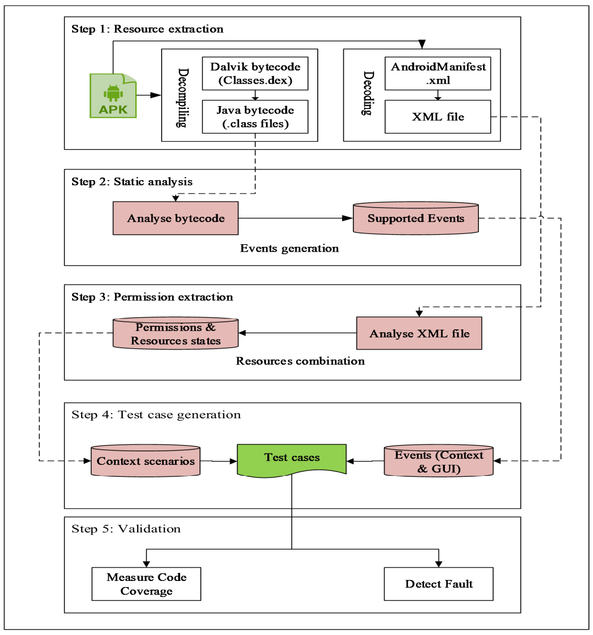

Fig. 1 Framework of TEGDroid

A. Step 1:Resource Extraction

Android apps are developed in Java and compiled to the dex file which runs on a special virtual machine called Dalvik Virtual Machine (DVM). They are zipped into a single application package file (APK) that is distributed in stores from where mobile device users can download to their devices. An APK is a compressed file format which contains various resources that comprises an Android application. It consists of the program code (dex file), Android `Manifest .xml` file and resources. To analyse the required resources such as bytecode and `manifest.xml`, the APK file needs to be de-compressed to extract the required resources. The outputs generated from this stage are the application's Dalvik Executable file (DEX) that contain the Java bytecode of an application and the AndroidManifest.xml file which contains all the permissions used by an app. The next step is to decompile DEX file and de-code the Android Manifest .xml for further analysis.

In order to obtain the bytecode and source code, reverse engineering is applied directly to the DEX file. This file is further de-compile to the Java bytecode using Dexpler [25] for the further analysis. The Java bytecode is used as input for the TEGDroid.

AndroidManifest.xml file is an encoded file format by the Android system. In order to extract app's permission, the extracted AndroidManifest.xml file needs to be de-coded to enable reading its information. The file is de-coded using apktool's decode function to enable reading the information of permission. The output of this stage is a readable AndroidManifest.xml file that contains declarations of permissions which an app must have in order to access protected resources.

B. *Step 2: Static Analysis*

Static program analysis is the analysis of software that is performed on the source code or bytecode without executing the programs to extract some information about the software [26, 27]. Program analysis is usually performed by a special automated tool that is designed to extract specific information. The static analysis targets the GUI events and the set of context events that can be detected vias the analysis of Intent message through the app's code.

In Android apps, there is the class which defines an Activity; the main app component that is responsible for presenting a GUI window that the user interacts with. The GUI widgets are linked to the listener object which is in turn connected to the eventHandlers that are responsible for executing user events. By analysing the GUI and event handlers, information about all the GUI events can be obtained. The context events (otherwise known as system events) are generated by the Android system in response to the device context [24]. Information about these events can be obtained by analysing the callback and Intent messaging system. The analysis performed Intent tracking to identify the system generated events by the device. The Intent messaging system is analysed to obtain information about the context events.

The control-flow analysis focuses on an essential aspects Android, which are the callbacks for a variety of interactions. It comprises lifecycle and interactions of user event-driven components that executes in the app's UI thread (the main thread). The analysis also targets activities, dialogs, and menus as the main components of an app. Two categories of callbacks: Lifecycle callbacks that manage the lifetime of app components and Lifecycle callbacks for activities, dialogs, and menus are considered in the analysis. The Lifecycle callbacks for activities, dialogs, and menus describe key changes to the observable state and to the possible run-time events and behaviour.

Static analysis is generally used by developers and researchers such as [27-29] to analyse software and extract information that can be used for software testing, software redocumentation and software understanding. There are several open source tools for static analysis of source code/ bytecode targeting different programming languages and platform. Examples are WALA framework used in [28] and GATOR used in [29]. This study employs static program analysis in GATOR [30] tool to obtain the vital information about an app that is required to identify supported events for test case generation, because it is designed specifically for java and android platform. GATOR is a Program Analysis Toolkit for Android that performs control flow analysis on application's GUIs, callback methods and Intent. A static control flow analysis is performed on the Java bytecode which tracks the callback methods and Intent messages. The control flow analysis is represented in form of a graph called Windows Transition Graph (WTG). The WTG represents both the events from the GUI and context events extracted through the analysis of callbacks and Intent messaging respectively. We used the WTG to obtain information about the supported events which is further used as input for test case generation. A fragment of the generated WTG is shown in Figure 2 below.

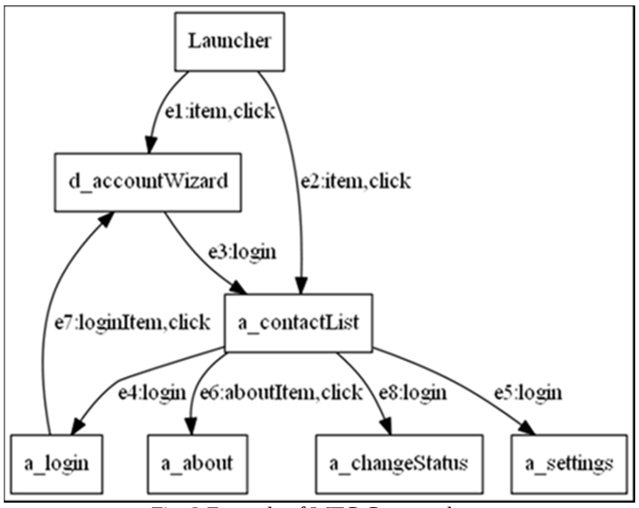

Fig. 2 Example of WTG Generated

The WTG comprises nodes and edges representing app's window/activity and events respectively. The window/activity comprises GUI and their properties which can be used to trigger an event on the app. In order to extract information (eS) about mobile app supported events from the WTG, a graph search algorithm is used on the WTG. Algorithm 1 was designed based on the concept of Dijkstra's algorithm that considers weighted edges to traverse the WTG.

The algorithm traverses the WTG from the start node to explore each path p in the graph as shown in Lines 3-5. Paths p in the graph represents an event that can be executed on an application where label assigned to the path indicates the sequence of execution. Based on this, a list of application's supported events is generated with their sequence of execution. The algorithm continues by backtracking of each p to identify the source node (Lines 6-7). In order to fire the events on an app, information about the events that includes handler method and GUI widgets is required. Therefore, the algorithm gets the GUI widgets contained in each source node and extracts their properties that are view ids and handlers and add the properties to the event summaries as in Lines 8-11.

At this point, the output is a summary of all events supported by an app. The event summary comprises an event id, the source of event, destination, event type and processing method. However, due to nature of static analysis it does not identify the dynamic behaviour of the resources (hardware) from which scenarios of context events are generated. This is main goal of the application of combination in order to generate the different states of resources.

```
Algorithm 1: Static Analysis
Input: AUT: App under test
Input: WTG: Windows transition graph (AUT)
Output: eS: Events Summaries
1   Procedure analyseApp(wtg)
2    eS ← GetEvnetSummaries(w,p)
3    for all paths P in graph do
4        while p ≠ explored then
5            Event e ← getPath(g)
6            for all e ∈ EventsSet do
7                sourceWindow W ← getSourceOfPath()
8                foreach w ∈ sourceWindow do
9                    eh ← getEventHandler()
10                   gw ← getGUIWidgets()
11                   id ← getProperties(GW)
12                   eS.add(e,id,gw,eh)
13               End
14           End
15       End
16   End
```

C. Step 3: Permission Extraction

The xml file contains all permissions assigned to an application. The permission list comprises all the resources e.g., camera, GPS that a mobile app may potentially require to run and the context events usually occur from the resources used by a mobile app [31]. Therefore, by analysing these permissions, the resources can be detected and subsequently the context events triggered by the resources can be identified.

The permissions assigned to an application are declared in a <uses permission> tag in the manifest file as shown in the example below:

```
<uses permission
Android:name="android.permission.INTERNET"
/>
```

The extracted output is a list of all permissions declared for the app. Table I shows a list of the permissions from `manifest.xml` for Beem app. Using the permission list generated for an app, the resources used by the app can be identified. The resources are then identified manually by considering what does the permission requires in order to run.

TABLE I
PERMISSION PROPERTIES

Permission	**Resource**	**State**
INTERNET	Radio, GPRS, Wifi	on/off on/off on/off
VIBRATE	Vibrator	on/off
WRITE_EXTERNAL_STORAGE	Sd card	Free/full
READ_EXTERNAL_STORAGE	Sd card	Free/full
ACCESS_NETWORK_STATE	Radio Receiver	on/off

In order to generate various scenarios of the executing context of an app, combination technique is applied on the resource conditions. For all resources that have links, the candidate states can be combined to generate the set of executing scenarios. This information is subsequently used to generate test cases that are capable of testing the context events. Based on the state of each resource, they are combined.

In order make sure that app functions correctly, there is need to have knowledge of the possible states of resource in order to know the different behaviours that can be triggered by the resource state [32]. This can help a tester to know the root cause of test failure during testing. The results of the combination are used to manually and add the conditions for the test scenarios to the test cases.

D. Step 4: Test Case Generation

In software engineering a test case is regarded as a set of conditions under which a tester will determine whether an application software system or one of its features is working as it was originally established for it to do. It may take many test cases to determine that a software program or system is considered sufficiently analyzed to be released.

There are several test generation techniques for test automation such as script based, random, capture/replay, search based and model based. Each technique uses a specific formalism of algorithm to process the application's information to generate test cases. In this study, search-based testing technique is utilized for generating test cases. Search-based testing technique help in reducing the costs of testing using the smallest set of test cases that can cover all the branches in a program. The event summaries (eS) extracted by Algorithm 1 is used as input for test case generation. Robotium testing framework is used to generate JUnit test cases that can be run test an app. The rule guiding the test case generation is defined as follows.

Rule: [Test generation]. For each event `e = e1, e2, e3, . . .` the event id `e(ei)` is translated to a matching Robotium API call which can trigger the event.

The description of the test case generation process is described in algorithm 2. The algorithm receives the eS as input for test case generation. Each event in the event summaries is a test path that can be translated to a test case. Beginning from the start state (event with id 1), the Algorithm extracts each event with the event id, widget and handler and translate it to a test case as shown in (Lines 2-4). The algorithm adds each generated test case to the list of test cases with the triggers in lines 6-9.

Algorithm 2: Generate TestCases

```
Input: AUT: App under test, eS: Event Summaries
Output: TC: Test Cases
1  Procedure Generate Test case()
2  testPath $P_t$ ← getEvent(eS)
3  while $P_t$ not empty do
4      testCase Tc ← genTestCase()
5      foreach Tc ϵ Tc set do
6          Wp ← getWidgetProperties()
7          W ← getWidget()
8          h ← getHandler()
9          Tc.add(e,wp,h)
10     End
11 End
```

The *eS* is used as input for the test generation algorithm to generate test cases for a given app by picking each event from the eS to generate sequence of test cases as described above. The output from the test generation algorithm is a set of test cases for each given app. However, script-based technique is used to manually write test condition for the test scenarios obtained from the resource combination and add it to the test cases generated. The scripted-based testing technique offers scripting languages which control an app programmatically that help a tester to write test scripts manually. It is done by giving a set of instructions that will be performed on the system under test to trigger events in order to test that the system functions as expected.

III. Results and Discussion

In order to measure the effectiveness of the TEGDroid, a case study was conducted on real-world Android application from GitHub and SourceForge. Eight (8) open source apps were selected across different categories such as, tools, communication, music and audio. These apps were used by other researchers for testing to evaluate their approach.

Experimental Setup. An Android device (real or emulated device) is needed in order to test a mobile app. In this research, an emulator was configured based on x86 Intel configuration with Android version 5.1 for the experiment. 1GB of memory was configured on the emulator to host the applications and logs. Specifically, Ubuntu 16.04 was used in running the experiments. The PC used is equipped with an i7 Intel processor with 8GB of RAM.

Benchmark. Table II presents list of the selected applications used for evaluation of TEGDroid. The apps are downloaded from GitHub and Sourceforge, which are popular benchmarks with different functionalities. The source lines of code of the apps ranges from 1.6K to 97K with average of 20.8K. With average of 1432 classes, 2458 methods and 14.4 activities.

TABLE II
BENCHMARKS OF THE APP'S USED FOR EVALUATION

App Name	SLOC	ELOC	Class	Method	Activity
Barcodescaner	6549	3477	110	565	9
Beem	21179	9123	227	1640	12
Marine Compass	1654	904	92	150	4
Open Camera	4862	2263	25	195	2
Pedometer	14654	6380	2208	11487	2
Subsonic	17779	8798	702	4602	13
TippyTipper	2284	1004	67	225	6
WordPress	97891	70596	8029	1309	68
Average	**20856**	**12818**	**1432**	**2458**	**14.5**

Two experiments were conducted to evaluate TEGDroid: code coverage analysis and mutation testing. Subsequent sections present discussion on the results of the experiment tal evaluation.

A. Coverage Result

This study has the main aim of checking the effectiveness of TEGDroid in terms of the coverage achieved on applications. Code coverage is used by several researchers and practitioners as a system of evaluating the effectiveness of testing techniques. Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test case runs [33, 34]. When the percentage measure of a program has high code coverage, it has had more of its source code executed during testing. This suggests it has a lower chance of containing undetected software bugs compared to a program with low code coverage.

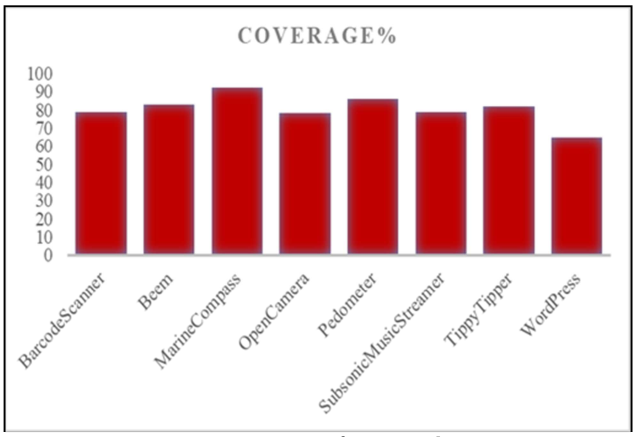

Fig. 3 Coverage of TEGDroid

There are several tools used for measuring code coverage for different programming languages such as jacoco and Emma. Emma code coverage library has now been built-in to the Android SDK which makes it easy to use for measuring the coverage during testing of Android mobile apps. This has made Emma [35] popular and is used by a variety of mobile apps testing techniques to measure the code coverage.

Figure 3 show the results obtained by the TEGDroid on the seven selected apps. Based on the results, TEGDroid has achieved LOC coverage of 79% for BarcodeScanner app, 83% for Beem app, 91% for MarineCompass, 78% for OpenCamera, 86% for Pedometer, 78% for Subsonic, 82% for TippyTipper and 65% for WordPress. TEGDroid achieved LOC coverage of 65% -91% across the eight (8) apps used.

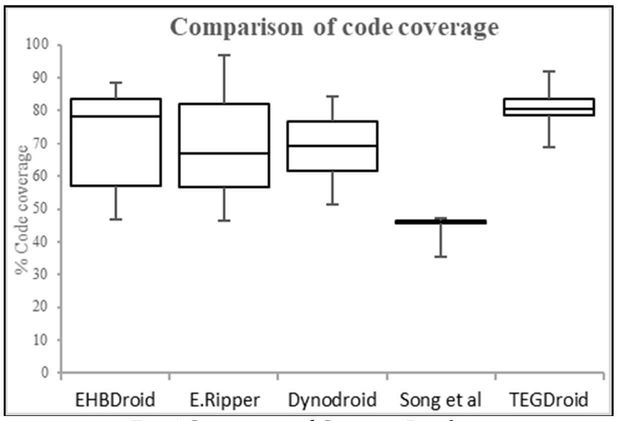

Fig. 4 Comparison of Coverage Results

We statistically performed a comparison of code coverage results of TEGDroid with other approaches as shown in Figure 4. The y-axis represents the code coverage result in percentage and x-axis shows the approaches used in the comparison. When we look at the spread of the coverage, it shows that the coverage achieved for most of the applications is 57% - 83% by EHBDroid, 57% - 82% by E.Ripper, 61% - 77% by Dynodroid, 45% - 47% by Song et al., TEGDroid obtained 79% - 84%. For the median of the coverage EHBDroid obtained 78.3%, E.Ripper 67%, Dynodroid 76.6%, Song et al. 46.5% and TEGDroid achieved 80.5. In comparison to the other approaches, TEGDroid obtains higher coverage above the median for most of the applications. This indicates that TEGDroid have higher code coverage compared to all the approaches.

B. Fault Detection using Mutation Testing

Code coverage has been recently criticized by many researchers [36-38] for validating the quality of software. Mutation testing changes a software artifact such as a program, requirements specification, or a configuration file to create new versions called mutants [39]. It is also called fault-based testing technique which measures the fault detection ability of a given test set by measuring the number of mutants being killed [40]. The general principle underlying mutation testing work is that the faults used by mutation testing represent the mistakes that programmers often make [41]. By Comparing mutation with other traditional testing technique like line coverage which only measures the percentage of code being executed, mutation testing could actually identify the ability of fault detection for tests [42]. The mutation analysis is represented as mutation scores (MS)

In this study, muDroid [40] is used for mutants generation and running mutation. MuDroid is an open source mutation testing technique for android apps that generates mutants based on six operators Arithmetic Operator Replacement (AOR), Inline Constant Replacement (ICR), Logical Connector Replacement (LCR), Negative Operator Inversion (NOI), Return Value Replacement (RVR) and Relational Operator Replacement (ROR). It performs an analysis during testing to determine if the generated mutants are alive or killed. The test cases generated from the TEGDroid are injected into the muDroid to run test on an app. During testing, when the test cases detect a mutant in an app, then the mutant is said to be killed. Once the mutants are killed, it shows that the generated test cases have the capability to reveal many faults in mobile apps.

muDroid generates several mutants in hundreds or thousands based on the size of a mobile app. To run thousands of mutants on an app, it requires a lot of resources and time [40]. To make the testing practical, muDroid employs several selection criteria which a tester can choose for the mutant selection. The random selection criterion is configured for mutant selection, because it is an efficient way to reduce the number of mutants by randomly selecting x% from the total mutants and it is the most common used cost reduction strategy [40]. Based on the experiment for the eight apps, MuDroid generates a total number of 34,275 mutants for the six mutation operators, as shown in Table III. Based on the criterion, some mutants were selected for each mutation operators.

Based on the results, all the selected mutants for the operators were killed with ROR having the highest mutation score of 2.402 for the total number of all the applications mutation score as shown in Table IV. A total of 6314 mutants were selected for the test for all the applications. Based on the results, all the selected mutants were killed with a mutation score MS=1.000 for each application. It can be concluded that the test sets generated from our approach have high mutant coverage, therefore it has the capability to reveal many faults in mobile apps.

IV. Conclusion

In this paper, we have presented an approach called TEGDroid for generating test case for Android Apps considering context and GUI Events. An experiment was performed on real world open source mobile apps to evaluate TEGDroid. Experimental result indicated that TEGDroid can significantly achieve high coverage as compared to other state-of-the-art approaches. We applied mutation testing to evaluate the fault detection capability of our approach. The mutation analysis result shows that 100% of the generated mutants were killed. This indicates that TEGDroid have the capability to detect faults in mobile apps.

TABLE III
MUTANTS GENERATED FOR EIGHT SELECTED APPS

Application	Mutants		Mutant operators					
	Generated	Selected	AOR	ICR	LCR	NOI	ROR	RVR
BarcodeScanner	5130	536	34	165	45	0	83	209
Beem	4949	401	48	118	81	0	59	95
MarineCompass	926	92	36	15	0	0	40	1
Open Camera	4664	320	95	55	8	0	157	5
Pedometer	9499	1476	378	439	0	0	578	84
SubsonicMusic	8118	665	77	160	121	0	112	195
TippyTipper	989	178	27	57	10	0	41	43
WordPress	17643	2646	410	222	169	0	1029	815
Total	**52918**	**6314**	**1105**	**1231**	**434**	**0**	**2099**	**1447**

TABLE IV
MUTATION SCORE

Application	Mutants		Mutation Score of the operators						MS
	Selected	Killed	AOR	ICR	LCR	NOI	ROR	RVR	
BarcodeScanner	536	536	0.063	0.307	0.084	0.000	0.154	0.389	1.000
Beem	401	401	0.119	0.294	0.202	0.000	0.147	0.236	1.000
MarineCompass	92	92	0.391	0.163	0.000	0.000	0.434	0.010	1.000
OpenCamera	320	320	0.297	0.172	0.025	0.000	0.491	0.016	1.000
Pedometer	1476	1476	0.256	0.297	0.000	0.000	0.389	0.056	1.000
SubsonicMusicStreamer	665	665	0.115	0.240	0.182	0.000	0.168	0.293	1.000
TippyTipper	178	178	0.152	0.320	0.056	0.000	0.230	0.242	1.000
WordPress	2646	2646	0.155	0.084	0.064	0.000	0.389	0.308	1.000
Total	**6314**	**6134**	**1.548**	**1.877**	**0.613**	**0.000**	**2.402**	**1.550**	**1.000**

ACKNOWLEDGMENT

We would like to acknowledge the support from Ministry of Higher Education and university Tun Hussein Onn Malaysia (UTHM) in undertaking the research under the Graduate Research Assistant for Postgraduate Research Grants (GPPS) and Fundamental Research Grant (FRGS) Vot number 1610.

REFERENCES

[1] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in 7th International Workshop on Automation of Software Test (AST), 2012, pp. 29-35.

[2] T. Tamilarasi and M. Prasanna, "Research and Development on Software Testing Techniques and Tools," in Encyclopedia of Information Science and Technology, Fourth Edition, ed: IGI Global, 2018, pp. 7503-7513.

[3] I.-A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, and A. Usman, "AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing," IEEE Access, vol. 7, pp. 17158-17173, 2019.

[4] I. Qasim, F. Azam, M. W. Anwar, H. Tufail, and T. Qasim, "Mobile User Interface Development Techniques: A Systematic Literature Review," in IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2018, pp. 1029-1034.

[5] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2011, pp. 252-261.

[6] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software," Automated Software Engineering, vol. 21, pp. 65-105, 2014.

[7] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in 2nd International Conference on Computer Engineering and Technology (ICCET), 2010, pp. V2-297-V2-301.

[8] I. C. Morgado, A. C. Paiva, and J. P. Faria, "Automated pattern-based testing of mobile applications," in 9th International Conference on the Quality of Information and Communications Technology (QUATIC), 2014, pp. 294-299.

[9] I. A. Salihu and R. Ibrahim, "Comparative Analysis of GUI Reverse Engineering Techniques," in Advanced Computer and Communication Engineering Technology, ed: Springer, 2016, pp. 295-305.

[10] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, et al., "An orchestrated survey of methodologies for automated software test case generation," Journal of Systems and Software, vol. 86, pp. 1978-2001, 2013.

[11] P. Aho, M. Suarez, A. Memon, and T. Kanstrén, "Making GUI Testing Practical: Bridging the Gaps," in 12th International Conference on Information Technology-New Generations (ITNG), 2015, pp. 439-444.

[12] D. Amalfitano, N. Amatucci, P. Tramontana, A. R. Fasolino, and A. M. Memon, "A General Framework for comparing Automatic Testing Techniques of Android Mobile Apps," Journal of Systems and Software, 2016.

[13] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," IEEE Transactions on Reliability, vol. 68, pp. 45-66, 2018.

[14] I. A. Salihu and R. Ibrahim, "Systematic Exploration of Android Apps' Events for Automated Testing," in Proceedings of the 14th International Conference on Advances in Mobile Computing and Multi Media, 2016, pp. 50-54.

[15] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 559-570.

[16] K. Song, A.-R. Han, S. Jeong, and S. D. Cha, "Generating various contexts from permissions for testing Android applications," in SEKE, 2015, pp. 87-92.

[17] A. Méndez-Porras, C. Quesada-López, and M. Jenkins, "Automated testing of mobile applications: a systematic map and review," in XVIII Ibero-American Conference on Software Engineering, Lima-Peru, 2015, pp. 195-208.

[18] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci, "Considering context events in event-based testing of mobile applications," in IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2013, pp. 126-133.

[19] W. Song, X. Qian, and J. Huang, "Ehbdroid: beyond GUI testing for android applications," in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 27-37.

[20] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 224-234.

[21] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," Journal of Systems and Software, vol. 83, pp. 60-66, 2010.

[22] T. Griebe and V. Gruhn, "A model-based approach to test automation for context-aware mobile applications," in Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014, pp. 420-427.

[23] T. A. Majchrzak and M. Schulte, "Context-dependent testing of applications for mobile devices," Open Journal of Web Technologies (OJWT), vol. 2, pp. 27-39, 2015.

[24] S. Yu and S. Takada, "Mobile application test case generation focusing on external events," in Proceedings of the 1st International Workshop on Mobile Development, 2016, pp. 41-42.

[25] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, 2012, pp. 27-38.

[26] B. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D. Marsh, "Industrial perspective on static analysis," Software Engineering Journal, vol. 10, pp. 69-75, 1995.

[27] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, et al., "Static window transition graphs for Android," Automated Software Engineering, vol. 25, pp. 833-873, 2018.

[28] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in International Conference on Fundamental Approaches to Software Engineering, 2013, pp. 250-265.

[29] I. A. Salihu, R. Ibrahim, and A. Mustapha, "A Hybrid Approach for Reverse Engineering GUI Model from Android Apps for Automated Testing," Journal of Telecommunication, Electronic and Computer Engineering (JTEC), vol. 9, pp. 45-49, 2017.

[30] "GATOR: Program Analysis Toolkit For Android."

[31] S. Mujahid, R. Abdalkareem, and E. Shihab, "Studying permission related issues in android wearable apps," in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 345-356.

[32] A. Usman, N. Ibrahim, and I. A. Salihu, "Test Case Generation from Android Mobile Applications Focusing on Context Events," in Proceedings of the 2018 7th International Conference on Software and Computer Applications, 2018, pp. 25-30.

[33] J. Levinson, Software Testing with Visual Studio 2010: Pearson Education, 2011.

[34] F. Horváth, T. Gergely, Á. Beszédes, D. Tengeri, G. Balogh, and T. Gyimóthy, "Code coverage differences of Java bytecode and source code instrumentation tools," Software Quality Journal, vol. 27, pp. 79-123, 2019.

[35] 2018 Emma, An open source Java code coverage tool [Online]. Available: http://emma.sourceforge.net/.

[36] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," IEEE Transactions on Software Engineering, vol. 41, pp. 803-819, 2015.

[37] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 435-445.

[38] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 72-82.

[39] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in Advances in Computers. vol. 112, ed: Elsevier, 2019, pp. 275-378.

[40] Y. Wei, "MuDroid: Mutation Testing for Android Apps," 2016.

[41] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," IEEE transactions on software engineering, vol. 37, pp. 649-678, 2011.

[42] C. Iida and S. Takada, "Reducing mutants with mutant killable precondition," in 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2017, pp. 128-133.