Mutation Testing Techniques for Android Applications: A Comparative Study

Ibrahim Anka Salihu
Department Software Engineering
Nile University of Nigeria
Abuja, Nigeria
https://orcid.org/0000-0001-7015-8741

Asma'u Usman
Department Computer Science
Abdu-Gusau Polytehcnic
Talata Mafara, Nigeria
asmee08@gmail.com

Ndukwe-Oke Eke
Department of Computer Science
Nile Univeristy of Nigeria
Abuja, Nigeria
ndukwe.eke@nileuniversity.edu.ng

Rosziati Ibrahim
Department Software Engineering
Universiti Tun Hussein Onn Malaysia
Batu Pahat, Malaysia
rosziati.ibrahim@uthm.edu.ng

Muhammad Bello Kusharki
Department of Information and Communication Technology
National Defence College
Abuja, Nigeria
https://orcid.org/0000-0003-0901-2326

2023 2nd International Conference on Multidisciplinary Engineering and Applied Science (ICMEAS) | 979-8-3503-5883-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICMEAS58693.2023.10429866

***Abstract*—Mobile applications are becoming increasingly used to achieve various computing needs. Hence, it is essential to guarantee quality of the applications. Software testing has been the main activity for ensuring the quality of software, however, it is a critical and costly activity. Furthermore, code coverage is commonly used as metric for verifying the effectiveness of testing technique. However, numerous researchers and experts claimed that considering code coverage only to ascertain the testing quality is not enough to ensure quality of apps. In view of this, mutation testing has been introduced to complement the procedure by assuring that apps behave as expected and are released free of faults. Several techniques/tools were proposed based on different syntax and mutation operators. This study presents a comparative study of six (6) mutation techniques/tools for Android applications to highlight their strengths and weakness which will give an insight to researchers on the directions for further research.**

Keywords—Android Application, Mobile Application, Mutation Testing, Software Testing

I. INTRODUCTION

The popularity of Smartphones has led to the development of mobile applications (mobile Apps) with advanced functionalities that enable them to be used in almost every facet of our daily lives, from communication, social networking and entertainment to education, health care, transportation e.t.c. [1][2][3]. Mobile apps are now advancing in capacity, structure and functionality as they are being used to address several computing needs [4]. Due to their usage in critical area of our lives, it is vital to ensure their quality [5]. To develop high-quality and more reliable mobile apps, there is a need for a comprehensive testing of the apps before release [6][7].

Software testing is the key activity for ensuring the quality software, however, it is a tedious and expensive activity in the lifecycle of mobile apps as a result of the increasing request for new apps and the quick evolution of mobile devices and frameworks [5][8]. Other challenges are due to the nature of mobile apps that allows them to handle rich innovative features apart from the conventional events such as swipes, pinches and events from sensors which are not directly linked to a particular GUI object [9][10]. These factors together bring difficulty in generating rich test cases that can explore different behaviours of applications [11].

Previous studies has proposed several techniques to deal with these challenges such as [4][12][13][14]. However, some of these challenges are yet to be resolved and many applications are released with numerous faults affecting their quality [15]. In order to provide improvement on the effectiveness of testing, several techniques focusing on test case generation for the various features of apps were proposed such as [16][11][17]. To verifying the testing technique's effectiveness, code coverage metric is commonly used [18]. Though, recently, numerous researchers and experts claimed that utilising code coverage only to ascertain the testing quality is not enough to ensure quality of apps [19][20].

Therefore, mutation testing (MT) have been introduced to complement the activity by making sure that apps behaves correctly and are delivered free of faults. The main goal of MT is to verify the fault detection capability of a given test-suite [19]. It is believed that any given test suite that kill the generated mutants has the ability to discover numerous faults in an app [19]. This study tries to explore some selected mutation testing techniques to identify the type of operators generated and the effectiveness of the techniques. The remainder of the article is as follows. Section II present the background and related works, section III present the methodology, section IV present the comparative analysis, section V presents the results and discussion and lastly section VI presents the conclusion.

II. BACKGROUND AND RELATED WORKS

Mutation testing (MT) is usually used as a means of assessing the completeness of test suites, for guiding the generation of test-cases for supporting experimentation. It is recently gaining recognition industry and mong researchers [8]. During mutation testing a software system is modified, for example, requirements specification, the program or configuration file, to produce new versions referred to as the mutants [19]. The mutants which are meant to be the defective versions are created through the application of certain rules for modifying the syntax of the artifact of software. The rules applied are called mutation operators (MOs). Mutation operators are designed to make changes that imitate

programmers mistakes and can further introduce changes that drive testers to design test inputs which may find faults [19][21]. The test suite is run against each mutant to measure its ability of killing the mutants. This is a measurement that is usually referred to mutation adequacy score. MT has generally been found stronger than other test criteria and usually a test that kills the mutants generated has the tendency of exposing numerous faults in a mobile app [22].

In the past few years researchers has proposed several mutation testing techniques/tools based on different mutation operators. These techniques have been practically applied to many languages, including Fortran [23], [20], C program [24] [25], Java [26][27], JavaScript [28], and web applications [29]. One of the earliest techniques for mutant generation for Android mobile apps was proposed by [22].

Other techniques are muJava [27], the mutation technique which was designed for desktop applications by Nilsson and Offutt [30]. Several techniques were also proposed for the mobile apps such as AndroidMutants [19] and muDroid [21]. These techniques were designed by adopting similar principles (operators designed to alter the app's code) similar to that for desktop apps [18].

Deng et al., [22], proposed a technique based on eight mutant operators which introduce faults in the important elements of the Android apps, such as, activity lifecycle, event handlers, intents, and XML files (e.g., GUI or permission files). muDroid [21] implemented six (6) operators that mutate app at the bytecode level by creating one APK for each mutant.

Andrews et al. [31] conducted a study to verify whether faults generated by mutant and humans seeded faults can be archetypal of true faults. The research confirmed that the mutants precisely selected may not be cheaper to detect as compared to the real faults, hence can offer a good clue on adequacy of test-suite, while faults seeded by human can probably yield underestimates.

III. Methodology

This section presents a discussion on how the study was conducting. Android apps form the subject of most of the recent research papers in the field of software, as such, this study focuses on mutation testing techniques for the android apps. To select our subjects, we performed a mini literature survey on the papers published within the last 10 years on the popular research databases and indexing system of Google scholar and Scopus. We considered these two databases as a good source because they cover all foremost publishers, e.g., IEEE, ACM, Elsevier, and Springer publications. The articles title, keywords and abstract were considered in the search using the range of 10 years, 2013 to date to identify the popular MT techniques/tools that were published in the literature. The search resulted in six tools, muDroid [21], DroidMutator [32], uDroid [33], Edroid [34], mDroid+ [35] and MutAPK [36].

IV. Selected Techniques/Tool

Our study focuses on mutation testing techniques for the Android mobile apps that available mainstream research databases. This section presents an overview of the six techniques/tools selected for the study.

A. muDroid

muDroid [21] proposed and implemented six (6) mutation operators for the Android apps which works at the "Smali code" level. These operators are ROR (Relational Operator Replacement), NOI (Negative Operator Inversion), RVR (Return Value Replacement), LCR (Logical Connector Replacement), ICR (Inline Constant Replacement), ICR (Inline Constant Replacement) and AOR (Arithmetic Operator Replacement).

Based on the operators, the tool creates hundreds of mutants from an app. muDroid provides a selection criterion aimed at reducing the sum of mutants utilised for the testing step. Their criterion of selection identifies the least set of mutation operators that will not affect the test effectiveness through the selection of representative mutants crosswise the diverse mutation operators.

B. DroidMutator

DroidMutator [32] is developed to deal with some challenges of the existing tools such as stillborn mutant. It employs type checking to minimise the creation of stillborn mutants such as, those that may lead to failed compilations.

DroidMutator implemented 2 types of mutation operators. Mutation operators specific to Java and other Android specific operators. Overall, it implemented a total of 32 MOs, out of which 28 of them were built from the current MOs, while 4 were new mutation operators they proposed based on their context.

C. uDroid

µDroid [33] is an energy-aware MT framework for Android. It was developed after conducting a detailed study that identified the regularly encountered Android apps energy defects. The authors call them energy anti-patterns. The tool implemented 50 mutation operators in consistent with the energy defect patterns identified and grouped them into 28 classes.

These classes of the mutation operators were grouped into 6 types, that further captured the commonalities among the diverse classes of operators. The Abstract Syntax Tree (AST) from the app's source code is used to searches for anti-patterns in AST which are encoded by mutation operators.

D. Edroid

Edroid [34] is a GUI based Android mutation testing tool with the primary purpose of mutating the main Android's components, e.g., activities, content providers, services and broadcast receivers by utilising the XML file's source-code. The tool implemented fourteen (14) mutation operators, which ten (10) among them were obtained from operators implemented in other fields similar to mutation testing for GUI, XML code, and from current research on Android mobile apps. Mutants are usually generated through the modification of the source files of an app either by using the mutation operators proposed or its subset.

E. MDroid+

MDroid+ [35] implemented a framework for MT of the Android mobile apps which seeks to empower developers in

writing test for mobile apps. It implemented 38 Android based and Java specific mutation operators which were constructed in accordance with the taxonomy of universal faults occurring in Android mobile apps which span across 14 different categories.

MDroid+ analyzes a target mobile app statically by parsing XML files in the case of resources or via the AST based Java code analysis.

F. MutAPK

MutAPK [36] is a mutation technique that enables the use of APK as input for mutant creation. It does not require source code to generate mutants, for the reason that the operations are done in the transitional representation of the code known as SMALI which require no compilation. It translated the 38 source code based MOs proposed by Linares-V´asquez et. al. [37] to its intermediate representation. However, since 3 of the operators mentioned previously did not produce results that is compilable, 35 out of the 38 operators were implemented.

V. THE COMPARATIVE STUDY

This section presents the details of the specific criterion we considered in the comparison followed by the results of our study.

A. Classification

According to Papadikis et. al. [8] mutation testing can be conceptually split into four major phases: 1) mutant generation; involves analysing classes for selecting a set of mutant operators and creating mutants by the instantiation of the selected mutant operators as actual executables, 2) Test selection; where tests are chosen to run alongside the mutants, 3) mutant insertion; where mutants are loaded into the execution environment, and 4) mutant detection; in which the chosen tests are run alongside the loaded mutant for identification. Table I presents the results of the classification.

TABLE I. TECHNIQUE/TOOL CLASSIFICATION

Mutation Technique	Mutant Generation	Test Selection	Mutant Insertion	Mutant Detection
MuDroid	√	-	√	√
DroidMutator	√	-	√	√
uDroid	√	-	√	-
EDroid	√	-	-	-
MDroid+	√	-	√	-
MutAPK	√	-	√	-

1) Mutant generation. All the techniques in the study generates mutants by selecting some set of defined operators as described in section II.

2) Test selection. In this regards all the techniques has automated test case generation with the exception of MuDroid as such non implemented test selection. In this case, MuDroid test suite is generated automatically but there is no mean of selected the test cases to run. As such, all generated test cases will be executed in testing process.

3) Mutant insertion. As mutants are generated, they need to be build to an APK to be able to instal and run a test on the mutant. According to the results, all the techniques surveyed has the capability to insert and compile mutants except EDroid. EDroid automated only the step of mutant generation while all other steps have to be done manually.

4) Mutant deletion. In order to verify if the test-suite has the capability to identify the mutants, a test is automatically run on the generated mutants. Based on our results, only MuDroid and DroidMutator has the ability of runnning a test automatically thereby enabling them to detect mutants and providing a report of the mutation analysis which will show the total number of all live and killed mutants.

B. Features

Mutation testing techniques are designed with various features. The three features considered in this study are 1) mutation level; which shows the level at which the techniques perform the mutant generation (source code or bytecode), 2) number of mutation operators used, and 3) fault detection model; which means concept used in creating the mutation operators to ensure that they can detect faults. Table II presents the features of the selected techniques/tools.

TABLE II. TECHNIQUES/TOOL FEATURES

Mutation Technique	Mutation Level	No. of Mutation Operators	Faults Model
MuDroid	APK	6	Uses Java variables and logical operators
DroidMutator	Source code	32	Uses Java reflection to identify object to mutate
uDroid	Source code	50	Energy-aware mutation operators
EDroid	Source code & XML	14	Adopted existing Java operators
MDroid+	Source code & XML	38	Specifically designed Android fault model
MutAPK	APK	35	Adopted potential fault profile (PFP)

1) Mutantion level. It is important to note that the source code of mobile apps is mostly not available, as such techniques that requires source code will be difficult to use. Based on the result of the anaylis, only MuDroid and MutAPK works with the APK of an app. EDroid and MDroid+ uses the source code and XML of the app, where as DroidMutator and uDroid works with the source code of app. Therefore, users has to find a way of extrating the source to enable them use these tools.

2) Number of mutation operators. Several research in mutation testing confirmed that one of the challenge of this type of testing is the cost of generating mutants and the time required to execute them. While some may consider using many operators to creating several mutants, others believed that may create time constrain. Our study shows that

MuDroid proposed six (6) mutation operators, DoidMutator 32, uDroid 50, EDroid 14, MDroid+ 38, MutAPK 35 operators.

3) Fault model used. It essential to determine a criterion for designing the mutation operators to ensure that mutants generated from such operators can cover most of the faults associated with an app. Our study shows that MuDroid considers Java variables and logical operators which they believe can cover many faults in Adroid. DroidMutator uses Java reflection to identify object to mutate which were considered as the operators. As uDroid is focused on identifying energy related faults, it uses energy-aware mutation operators. EDroid adopted existing Java operators proposed by other researchers and introduced three more. Mdroid+ used operators derived from manually analysing numerous software artifacts and MutAPK Adopted potential fault profile (PFP) proposed by other researchers. In all the techniques studied non of them device a means of designing mutation operators that considers the various categories of context events generated by by mobile apps, such as events from sensors and other sources.

VI. Conclusion

Recently mutation testing is gaining popularity from both industry and researchers as a fault-based testing that will deal with the shortcomings of code coverage in ascertaining the quality of software app. In this study we surveyed and compared six popular Android apps testing techniques/tools with intention of providing researchers and practitioners with the challenges associated with each in terms of cost and time to generate and run test.

Considering the evolution of Android device and it's mobile apps, there is a deficiency of techniques with mutation operators that can mutate the complex event types events handled by android mobile apps such as context event from internal and external sources. In this regard, future research needs to address other specific types of faults related with this kind of events. Techniques should offer support for mutant generation and execution for context events as well as to reduce the cost associated with mutation testing. Furthermore, it is important to investigate the impact of using large number of mutation operators in ensuring the quality of test suite and its impact on the time to generate the mutants.

References

[1] F. Nayebi, J. M. Desharnais, and A. Abran, "The state of the art of mobile application usability evaluation," *2012 25th IEEE Can. Conf. Electr. Comput. Eng. Vis. a Greener Futur. CCECE 2012*, pp. 1–4, 2012, doi: 10.1109/CCECE.2012.6334930.

[2] I. A. Salihu and R. Ibrahim, "Systematic exploration of android apps' events for automated testing," *ACM Int. Conf. Proceeding Ser.*, pp. 50–54, 2016, doi: 10.1145/3007120.3011072.

[3] I. A. Salihu and R. Ibrahim, *Comparative Analysis of GUI Reverse Engineering Techniques.*, Lecture No., vol. 362. Springer, 2016.

[4] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," *Proc. - Int. Conf. Softw. Eng.*, vol. 1, pp. 89–99, 2015, doi: 10.1109/ICSE.2015.31.

[5] P. Tramontana, D. Amalfitano, N. Amatucci, and A. R. Fasolino, "Automated functional testing of mobile applications: a systematic mapping study," *Softw. Qual. J.*, vol. 27, no. 1, pp. 149–201, 2019, doi: 10.1007/s11219-018-9418-6.

[6] L. Osterweil, "Strategic directions in software quality," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 738–750, 1996, doi: 10.1145/242223.242288.

[7] I. A. Salihu, R. Ibrahim, and A. Usman, "A Static-dynamic Approach for UI Model Generation for Mobile Applications," *2018 7th Int. Conf. Reliab. Infocom Technol. Optim. Trends Futur. Dir. ICRITO 2018*, pp. 96–100, 2018, doi: 10.1109/ICRITO.2018.8748410.

[8] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," *Adv. Comput.*, vol. 112, pp. 275–378, 2019, doi: 10.1016/bs.adcom.2018.03.015.

[9] A. Usman, N. Ibrahim, and I. A. Salihu, "Test case generation from android mobile applications focusing on context events," *ACM Int. Conf. Proceeding Ser.*, pp. 25–30, 2018, doi: 10.1145/3185089.3185099.

[10] I. A. Salihu, R. Ibrahim, and A. Mustapha, "A hybrid approach for reverse engineering GUI model from android apps for automated testing," *J. Telecommun. Electron. Comput. Eng.*, vol. 9, no. 3-3 Special Issue, pp. 45–49, 2017.

[11] A. Usman, N. Ibrahim, and I. A. Salihu, "TEGDroid: Test case generation approach for android apps considering context and GUI events," *Int. J. Adv. Sci. Eng. Inf. Technol.*, vol. 10, no. 1, pp. 16–23, 2020, doi: 10.18517/ijaseit.10.1.10194.

[12] I. A. Salihu and R. Ibrahim, "Reverse engineering mobile apps for model generation using a hybrid approach," *J. Telecommun. Electron. Comput. Eng.*, vol. 8, no. 4, pp. 1–5, 2016.

[13] A. Nazish, "A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications Summarized by : Nazish Asad," *Bibliometr. Data Bibliometr.*, pp. 1–2, 2012.

[14] A. Usman, N. Ibrahim, and I. A. Salihu, "Comparative Study of Mobile Applications Testing Techniques for Context Events," *Adv. Sci. Lett.*, vol. 24, no. 10, pp. 7305–7310, 2018, doi: 10.1166/asl.2018.12933.

[15] D. R. De Almeida, P. D. L. MacHado, and W. L. Andrade, "Context-Aware Android Applications Testing," *ACM Int. Conf. Proceeding Ser.*, pp. 283–292, 2020, doi: 10.1145/3422392.3422405.

[16] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Autom. Softw. Eng.*, vol. 21, no. 1, pp. 65–105, 2014, doi: 10.1007/s10515-013-0128-9.

[17] 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, doi: 10.1109/ACCESS.2019.2895504.

[18] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing Android apps," *Inf. Softw. Technol.*, vol. 81, pp. 154–168, 2017, doi: 10.1016/j.infsof.2016.04.012.

[19] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," *ICSE 2014 Proc. 36th Int. Conf. Softw. Eng.*, pp. 9853–9853, 2010, [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2568225.2568271.

[20] Y. Wei, "MuDroid: Mutation Testing for Android Apps Undergraduate Final Year Individual Project," 2016.

[21] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of Android apps," *2015 IEEE 8th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2015 - Proc.*, 2015, doi: 10.1109/ICSTW.2015.7107450.

[22] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Softw. Pract. Exp.*, vol. 21, no. 7, pp. 685–718, 1991, doi: 10.1002/spe.4380210704.

[23] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, 1991, doi: 10.1109/32.92910.

[24] M. E. Delamaro and J. C. Maldonado, "Proteum - A Tool for the Assessment of Test Adequacy for C Programs," no. 1, pp. 1–14, 1996.

[25] A. K. Elmagarmid, *MUTATION TESTING FOR THE NEW CENTURY MUTATION 2000 The Kluwer International Series on*. 2000.

[26] Y. S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, 2005, doi: 10.1002/stvr.308.

[27] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," *Proc. - IEEE 6th Int. Conf. Softw. Testing, Verif. Validation, ICST 2013*, pp. 74–83, 2013, doi: 10.1109/ICST.2013.23.

[28] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," *ICSTW 2010 - 3rd Int. Conf. Softw. Testing, Verif. Valid. Work.*, pp. 132–141, 2010, doi: 10.1109/ICSTW.2010.38.

[29] R. Nilsson, J. Offutt, and J. Mellin, "Test Case Generation for Mutation-based Testing of Timeliness," *Electron. Notes Theor. Comput. Sci.*, vol. 164, no. 4 SPEC. ISS., pp. 97–114, 2006, doi: 10.1016/j.entcs.2006.10.010.

[30] J. H. Andrews, L. C. Briand, Y. Labiche, A. S. Namin, and D. Seo, "Using Mutation Analysis for Assessing and Comparing Test Coverage Criteria Introduction Related Work Experimental Description Analysis Results Conclusion," no. Tse 2006, pp. 1–26, 2013.

[31] J. Liu, X. Xiao, L. Xu, L. Dou, and A. Podgurski, "DroidMutator: An Effective Mutation Analysis Tool for Android Applications," *Proc. - 2020 ACM/IEEE 42nd Int. Conf. Softw. Eng. Companion, ICSE-Companion 2020*, pp. 77–80, 2020, doi: 10.1145/3377812.3382134.

[32] R. Jabbarvand and S. Malek, "µDroid: An Energy-Aware Mutation Testing Framework for Android," *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, vol. 2017-Janua, pp. 208–219, 2017, doi: 10.1145/3106237.3106244.

[33] E. Luna and O. El Ariss, "Edroid: A mutation tool for android apps," *Proc. - 2018 6th Int. Conf. Softw. Eng. Res. Innov. CONISOFT 2018*, pp. 99–108, 2019, doi: 10.1109/CONISOFT.2018.8645883.

[34] K. Moran *et al.*, "MDroid+," pp. 33–36, 2018, doi: 10.1145/3183440.3183492.

[35] C. Escobar-Velasquez, M. Osorio-Riano, and M. Linares-Vasquez, "MutAPK: Source-codeless mutant generation for android apps," *Proc. - 2019 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2019*, pp. 1090–1093, 2019, doi: 10.1109/ASE.2019.00109.

[36] M. Linares-Vásquez *et al.*, "Enabling mutation testing for android apps," *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, vol. Part F1301, pp. 233–244, 2017, doi: 10.1145/3106237.3106275.