An Innovative Testing Model Using Gene Mutation Technique for Android Applications

Muhammad Arshad Javed, Dr. Rosziati Ibrahim

Department of Software Engineering, Faculty of Computer Science and Information Technology (FSKTM) UTHM, Malysia.

Android applications have more new updates, sales, and downloads than any other mobile platform popularity of Android smartphones. These programs' enormous amount of code raises serious questions about the software's quality. Testing Android applications, nevertheless, owing to the distinctive program structure and new aspects of evaluating modern Java programs, Simple test coverage metrics like statement coverage are insufficient to guarantee excellent quality in apps. applications for Android. While academics are very interested in developing better Android testing techniques, there are yet no practical methods for analyzing their suggested test selection tactics. We predict that mutation analysis, which has been proven to be a successful method of designing tests in other software domains, is also a workable option for Android apps. This research suggests a novel mutation analysis strategy tailored to Android app development. We propose mutation variant operators particular to the features of Android apps, such as the inherent event-drivenness, the specialized Activity lifecycle structure, and the widespread usage of XML files to determine layout and behavior. We also discuss an empirical investigation we conducted to assess these variant operators. We have created a program that modifies the source code of Android apps using the innovative Android mutation operators, producing variants that can be deployed and used on Android platforms. Through an empirical investigation of real-world apps, we assessed the effectiveness of Android mutation testing. This work offers a large empirical investigation using real-world applications, introduces various unique variant operators derived from a error analysis of Android applications, and draws conclusions after analysing the findings. The findings demonstrate that the innovative Android variant techniques provide comprehensive testing for Android apps. We also highlight difficulties, opportunities, and future research areas to improve the performance of variant analysis for mobile apps since the application of mutation testing to Android apps is still in its infancy.

Keywords: Mutation, Android application, Mutant operators, Machine learning.

1. Introduction

A piece of software known as a mobile application runs on mobility devices, such as a tablet or smartphone. As new base become accessible, additional applications are promoted, costs

decline, and many consumers go for many modern devices. Hence the number of mobile applications (apps) is increasing day by day. In the third quarter of 2014, Android accounted for 83.1% of market sales, while iOS came in second with 12.7%. [1]. The Google Play Store, the leading Android app marketplace, has over a million applications accessible for consumers, and hundreds more are uploaded every day. Quality is a critical and expanding issue, as might be expected. Numerous programs that are released onto the market have serious flaws, which frequently cause failures while being used. Bhattacharya et al. [3] examined 29,233 bug complaints in twenty-four popular open-access applications Android applications to examine the prevalence of software flaws in applications for Android. They discovered that over 8,500 bug reports were subsequently confirmed as issues by developers and subsequently addressed. They found issues with all of the Android apps they examined.

While inadequate use of software engineering concepts, such as minimal or no testing strategies, contributes to the problem in some cases, there is also a significant technological issue. Our entire research effort is to provide testing methodologies that can help programmers detect bugs in Android apps before they are released, especially in the code that uses new programming capabilities (as described in Section 2). In particular, we suggest making use of mutation analysis, a sophisticated testing method well-known for assisting engineers in creating effective tests. In order to create new mutation operators, we first analyse the distinct technological properties of Android apps. It is probable that tests that eliminate such mutations may find numerous use errors. The study has implemented both new Android variant operators and several outdated variant operators in a prototype mutation analysis tool developed and implemented. There are three methods to employ our Android mutation analysis tool. Mutation analysis stands among the highly effective approaches or generating test cases. Thus, extremely effective tests may be created using mutation [4-5]. Second, a mutation analysis tool may be used to compare various Android app testing strategies once it has been finished, refined, and made accessible to other researchers. Third, many pre-existing tests that a tester has are likely to be redundant. The following contributions are made by the paper:

- It describes brand-new mutation operators that are exclusive to Android applications.
- Eight Android applications are used to test these mutation operators.
- It pinpoints potential study areas for Android app mutation analysis.

The essay is structured as follows: Section 2 provides background information on the essential features necessary for framework development. Section 3 outlines the proposed approach, while Section 4 covers the experimental analyses conducted to evaluate the proposed models. Section 5 offers a concise summary and discusses future directions.

2. Background

Compared to traditional software, Android applications are developed differently and employ fresh control and data links. In this study, mutation testing—an established testing method—is being applied to a novel kind of software—mobile applications. We must thus

give a quick explanation of how an Android app functions before moving on to our study, followed by a description of mutation testing [6-8].

a) Programming Android Applications

The Android Application Development Framework is the development environment included with Android (ADF). An API is available through the Android ADF to assist with app development, GUI design, and device data access. Linux-based middleware, pre-installed programs, and system libraries are all part of the Android operating system [9]. Before version 4.4, Android ran Java apps using the Dalvik Virtual Machine [35]. (KitKat). Android Runtime took the role of Dalvik in the most recent version, Android 5.0 (Lollipop) (ART)[25]. However, according to Google, the majority of Dalvik-optimized programs ought to function under ART without any modifications [10, 11, 17].

The general design or coding process of Android apps are unaffected by the change. Android apps can broadcast its characteristics for use by other apps, but with some restrictions. An obligatory manifest file, four different sorts of components, and a new framework are all used in the construction of Android apps. The ADF retrieves information about the app from XML-based manifest files, containing configuration details and descriptions of the app's components[33].

With some restrictions, Android apps can also expose their attributes for use by other applications. Activities, Services, Broadcast Receivers, and Content Providers are the four different sorts of parts that make up Android applications. Based on one or more layout designs, an activity shows the user a screen. Different configurations for various screen sizes may be included in these layouts.

View widgets, or GUI controls, are defined by the layouts. A unique identity for each widget serves as the description of the controls and their layout in an XML configuration file. On the device, background service components are always active. They carry out actions like step tracking, keeping track of set alarms, and playing music that don't require user participation. Despite the fact that they might communicate with an action, which in turn communicates with the monitor, services do not interface with the screen directly. Calendar, pictures, contacts, and audio files are just a few examples of the structured data that a content provider keeps and makes available for users to access. Finally, a Broadcast Receiver manages announcements sent to the entire system, including low battery. An intent message, which contains both the information the component requires and the action the component should perform, is used to activate an Android component. Dynamic linkingof messages is supported by Android. Instead of being openly present in the app, calls are routed through the Android messaging service to allow this. All significant parts of Android must follow a predefined lifetime, including Services and Activities [12]. The ADF controls these actions. The lifespan of an activity is depicted in Fig. 1 as a series of occasions and conditions. Running, pausing, and stopping are the three states. Events onCreate(), onStart(), and onResume() bring about the Running state (). The Activity is put into the Paused state by onPause(), then into the Stopped state by onStop(), and finally back into the Running state by onResume(). The Activity can quit with a onDestroy() event or transition from Stopped to Running using onRestart(), onStart(), or onResume(). According to a subsequent explanation, ADF contacts lifecycle event handlers and is crucial to our research.

b) Mutation Analysis

This study uses mutation concepts to provide efficient tests for Android app parts. A software artifact is altered during mutation testing to produce new iterations known as mutants, such as programs, requirements specifications, or configuration files [13]. By applying criteria for altering the software artifact's syntax, mutants are often designed to be flawed versions. They are known as variant manipulators. The tester then sets tests, known as killing the mutant, that cause the original and each altered variant to display distinct characteristics. For instance, the ROR operator for obsolete development tools substitutes every occurrence of each relational manipulator, such as =, with remaining comparison operators, such as, ==, >, >=, and! =, as well as trueOp and falseOp, which determine conditions as true or false [14]. In certain cases, mutation operators produce modifications that are comparable to those seen in Activity Lifetime in Android applications, can occasionally be introduced, forcing testers to provide test data that reveals erros.

In certain cases, variant manipulators, produce modifications that are comparable to those seen in Activity Lifetime in Android applications, can occasionally be introduced, forcing the testing operators to provide sample data that are likely to uncover flaws. To determine the proportion of mutations that the tests successfully eliminate, each variant is tested against the performance in a test suite. The mutation adequacy score is what we refer to as. It has been consistently found that mutation testing is generally more rigorous than alternative evaluation methods. The prime reason for its robustness is the fact that it applies global requirements as well as local ones, such as the ability to traverse a sub path in the control flow graph or reach a statement (reachability) [14–16]. It also stipulates that the modified statement must produce a fault in the service's effective execution (infection), and that this error must spread to cause wrong external behaviour of the altered program (propagation from that mistaken state). Some mutants cannot be eliminated because they behave exactly like the original software regardless of the input. Equivalent describes these mutations. A notable challenge in variant testing is identifying and excluding similar mutations, which can be costly. Due to the alteration making the program syntactically wrong, certain mutants do not compile and are stillborn [14,17-19]. While most of these stillborn mutants may be prevented provided the variant manipulators are correctly created and executed, some do happen. A mutation system has to be capable of identifying stillborn mutants and excluding them from further analysis. C, Java, and Fortran are only a few of the numerous languages for which mutation operators have been developed [20–23]. Android app mutation operators concentrate on Android's unique characteristics, such as the manifest file, activities, and services.

3. Proposed Study

Figure 1 illustrates various modules which constitutes the proposed framework. The first module collects and organizes the applications. The second module is the pre-processing module, which initially checks and filters the actual java and XML based applications. The pre-processing module checks for the completeness of the applications and cross checks whether the applications can be compiled and executed successfully. Next comes the mutation module, where the java based and XML based application formats are muted and

fused to form a generalized APK file formats which are used as input to the proposed testing model. Followed by mutation process the suggested testing model operates to parse the APK based applications and compares each token of the statements in the source code of the application with the predefined application structure formats to verify whether the structure of the application is proper. Testing applications having similar instruction set as that of predefined instruction set are rejected, while unlike instruction set are accepted. The final stage in the framework is the evaluation phase which is used to evaluate the suggested test model using various metrics like accuracy, efficiency and quality [20]Proposed framework shown in fig. 1.

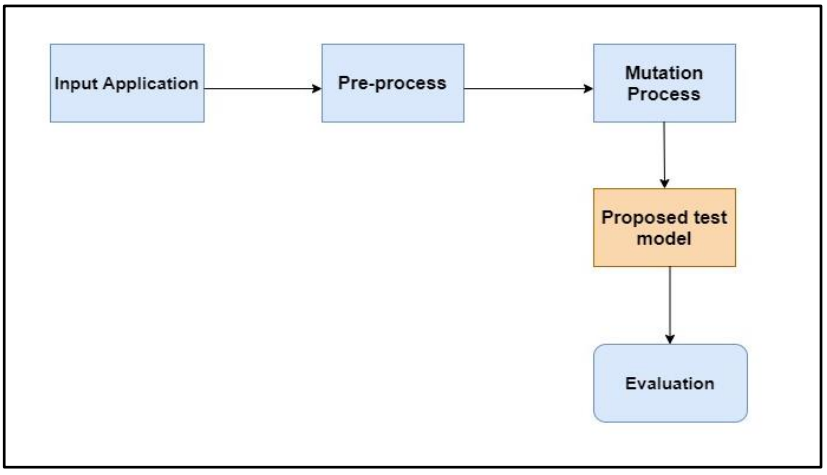


Figure 1. Proposed framework

The detailed execution process of the proposed test model for testing Android-based applications is depicted in Figure 2. This research utilizes various Android mutation analysis tools, including muDroid, which introduces new Android mutation operators alongside traditional Java mutation operators from MuJava and deletion operators. The analysis tool extends a portion of the MuJava mutant generation algorithm to accommodate these mutation operators for Android. It applies these mutation operators to the original Java files based on mutation principles and converts them to bytecode for standard Java mutation operators. XML-related mutation methods are applied to XML files, generating new versions of each file with XML mutants. Subsequently, in the creation of APK files, the Android mutation testing tool replaces these files to prepare them for runtime binding.

The mutation process involves selecting a modified Java bytecode-class file, integrating it with other proposed files, and generating a modified APK file that acts as a altered version of the Android app under examination. Some mutations may lead to compilation errors. Furthermore, several external Android automation tools, such as Robotium, Espresso, and Selendroid, are commonly utilized by researchers and developers to automate testing of Android apps. These testing frameworks are also adapted for use in Android mutation testing. Researchers can develop test cases using these Android automation frameworks designed to detect mutations. Once mutants are created and packaged into APK files, the system installs the test app on devices. It compiles all test cases, executes them across

various applications, and records the results [21–22].

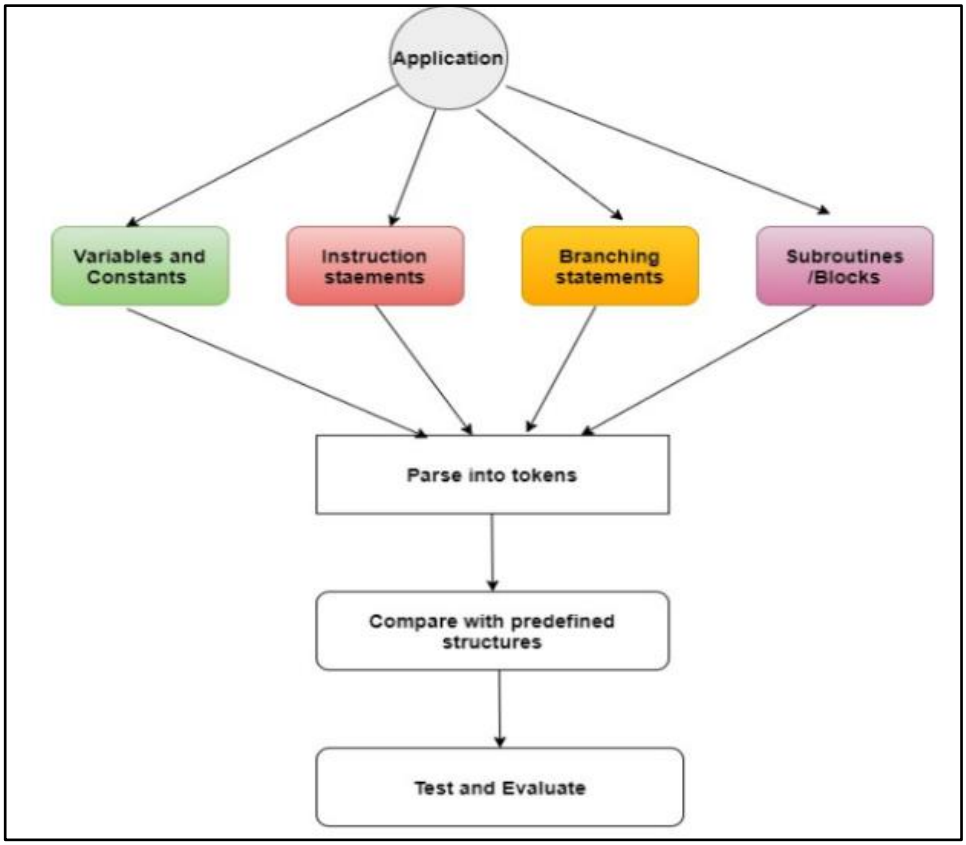


Figure 2. Overview of proposed testing model

After that, the Android device is given the APK mutant file. The mutation system executes all difficult instances in accordance with the mutants and records the results as actual outcomes. The Android research tool then compares the actual findings to the expected results after gathering all the results. If the actual test results differ from the anticipated test results, the mutant is documented as having been eliminated by the test. The Android mutation research tool calculates the mutation sufficient result at the conclusion, which is the ratio of mutants killed by tests to the total number of non-alike mutants. The researcher must manually eliminate equivalent mutants because the tool doesn't use any heuristics to help identify equivalent mutations [24].

a) Proposed Method to Reduce Mutants

In order to efficiently analyse the Android application instructions, we must first create a mutant branch. Construction of mutated buds and development of an innovative application constitute the initial stage of our technique, which converts the gaps identified of eradicating mutation into that of concealing mutant parts. P represents the initial program. Assume that statement s is the mutated statement and that statement s represents the mutated state. (P) becomes a mutant, denoted as (m), when (s) is swapped out for (s).

During testing a recessive variant, if a test datum eliminates m, it should approach (s) and yield a neighbouring state after performing s, i.e., s! = s. The marked statements, which indicate that m is eliminated, are utilized as actual outlets., represented as b, to produce a conditional declaration with ("s! = s") as the predicate. Thus, hiding the true fraction of (b) becomes the issue at hand while trying to eliminate (m). Due to the mutual mapping amid (b) and (s), or alternatively between (b) and (m), (B) is referred to as a recessive outlet. We merge all the variant divisions using both the original and modified statements after smearing all the variant tokens to P. The merged mutation stems are referred to as b1, b_2, b_N and their associated mutants as m_1, m_2, m_N, when (N) is the count of mutation descendant (mutants). All of the variant outlets are successively merged into the original applications to produce a new one, denoted as (P). Table 1 details the creation of mutation stems using MuClipse's common mutation operators. Here, MuClipse is a MuJava connector for Eclipse that has the ability to create and run variants dynamically. Mutation testing software like MuJava is frequently utilized in academic settings. This table shows the initial statement s, the updated phrase s obtained by applying a modified form to s, and the mutation stem b formed by fusing (s) and (s). For instance, after running AORB on "a_1 + b_1" a mutated statement "a b" is generated, and "a_1 + b_1" and "a b" are merged to form their matching mutant branch, If ((a_1 + b_1)! = (a_1 b_1))". Several of the mutation operators in Table 1 may have an effect on the execution result P. As an illustration, the results of applying AOIS on the variable "a" in the phrase "a_1 + b_1" are "+ + a_1", "a_1", "a_1 + +," and "a_1." According to our study, the words "+ + a_1" and "a_1" are instantly replaced with the words "(a_1 + 1)" and "(a_1 1)," respectively. We substitute "(a_1 + 1)" and "(a_1 1)," respectively, because "a_1 + +" and "a_1" affect the subsequent code in P [37].

Ansgle is a popular example of software testing [24,30,31], and Fig. 3(d) displays its Java version. Fig. 3 illustrates the creation of P as follows: (a) the code for P (Triangle); (b) the code from lines 7 to 9 in (a); (c) the mutant branches produced as a result of performing ROR on If (a_1 == b_1)" and AORB on "ans = ans + 1," respectively; and (d) the CFG following the insertion of mutant branches into P. The implanted mutant branches are shown in Fig. 3(d) by the dashed boxes. Although each contains two branches the actual branch and the fake branch, we only take the true branch into account. Additionally, the phrase (s) in the actual stem reveal that its original section is covered, i.e., weak mutation testing reveals an unexpected state. After the development of the novel application, P, the issue of deleting (N_n) mutants in (P) is turned into the issue of concealing N mutation nodes in (P). But the complexity of addressing the transformed problem is significantly increased by the abundance of mutant branches in (P). We demonstrate the increasing structural complexity in (P) using the software in ig. 3(a). To modify the program, all 15 of MuClipse's typical operators are chosen. From lines 6 to 15 in Fig. 3, 78 mutants are produced (a). Table 2 contains a list of the mutants, where "others" stands for the other nine mutation operators that produce no mutants. 26 mutants are produced for If (a_1 == b_1)" (lines 7 to 9 in Fig. 3 (a), with 16 of them coming from If (a_1 == b_1)" (line 7) and 10 from "ans = ans + 1." (line 8). There are additional 26 created mutant branches, and Table 3 lists their predicates. By incorporating these variant branches into the actual application, the simplified CFG of the new program is illustrated in Figure 3 (b) (lines 7 to 9), where each number corresponds to a predicate in Table 2.[26][27]

Table 1. The construction process of mutant branches

Mutation Operator	Description	Original Statement (s)	Muted Statement(s)	Mutant Branch (b)
MORB	Replacing the Fundamental Arithmetic Units	$c_1 + d_1$	$c_1 * d_1$	if ($(c_1 + d_1)$!= $(c_1 * d_1)$)
MORS	Instant Numerical Procedure Replacement	$c_1 + + + d_1$	$c_1 - - + d_1$	if (($c_1 + 1 + d_1$) != $(c_1 - 1 + d_1)$)
MOIU	Add minor arithmetic computations	$c_1 + d_1$	$c_1 + d_1$	if (($c_1 + d_1$) != $(-c_1 + d_1)$)
MOIS	Insertion of a short mathematical unit	$c_1 + d_1$	$+ + c_1 + d_1$	if (($c_1 + d_1$)! = $(c_1 + 1 + d_1)$)
MODU	Elimination of pointless unary arithmetic elements	$-c_1 + d_1$	$c_1 + d_1$	if (($-c_1 + d_1$) != $(c_1 + d_1)$)
MODS	Elimination of rapid arithmetic elements	$c_1 + + + d_1$	$c_1 + d_1$	if (($c_1 + 1 + d_1$) != $(c_1 + d_1)$)
RepOR	Alternative to Logical Activity	$c_1 > d_1$	$c_1 <= d_1$	if (($c_1 > d_1$) !=($c_1 <= d_1$))
RepDF	Substitution of Reliant Unit	$c_1 > d_1$ ||c < d	$c_1 > d_1$&& c < d	if (($c_1 > d_1$||$c_1 < d_1$)! =($c_1 > d_1$&&c < d))
RemDF	Effectively Removing Reliant Function	$!(c_1 > d_1)$	$c_1 > d_1$	if (!($c_1 > d_1$) != ($c_1 > d_1$))
AddDF	A reliant function is included	$c_1 > d_1$	$c_1 > d_1$)	if (($c_1 > d_1$)! = !($c_1 > d_1$))
ShifRO	Substitution for Shift Activator	$c_1 >> d_1$	$c_1 >>> d_1$	if ((c >>d)! = ($c_1 >>> d_1$))
LogR	Substitution for Logical Operator	$c_1 \| d_1$	$c_1 \& d_1$	if(($c_1 \| d_1$) != (c&d))
LogI	Inclusion of Logical Operators	$c_1 + d_1$	$\sim c_1 + d_1$	if(($c_1 + d_1$) !=($\sim c_1 + d_1$))

The variant outlets produced by AOIS in Table 2 require further conversion, as shown in Table 1. If the effect of a modified statement cannot be communicated or there is no reference to a variable in the subsequent code, the corresponding mutation is considered equal. Because of this, claims like "ans = ans + + + 1" and "ans = ans + 1"—which correspond to predicates 25 or 26 in Table 2—are illogical. Because "ans" starts off with a value of 0, further condition 7 in Table 2 is similarly impossible [28].

When 78 mutated variants are introduced into P, spanning lines 6 to 15 in Fig. 3(d), the number of paths increases 26-fold compared to the three branches in the original program., which are shown in Fig. 3(d). This suggests that lowering mutation stems in P is required because the line count in (P) is increased by at least 23 times, and each variant outlet comprises at least three lines of code (20 to 26). (P) must be simplified by determining the leading link among variant outlets. [29][30].

```
1.  Public static int getMut(int c, int b, int
    c){
2.  int ans;
3.  If (c<=0||d<=0||e<=0)
4.  return 4;}
5.  ans=0;
6.  If(c==d){
7.  ans=ans+1;
8.  }
9.  If (c==e){
10. ans=ans+2;
11. If (d==e){
12. ans=ans+3;
13. }
14. If(ans==0){
15. If(c+d<=e) ||(d+e)<=c||(c+e)<=d){
16. return 4;
17. } else {
    return 1;
18. }
19. }
20. If/(ans>3){
21. return 3;
22. } elseif (ans==1 &&c+d>c){
23. return 2;
24. } else if(ans==2 && c=e>d)
25. return 2;
26. } else if (ans==3 &&d+e>c){
27. return 2;
28. }
29. return 4;
30. }
```

Figure 3(a). Process of forming new program

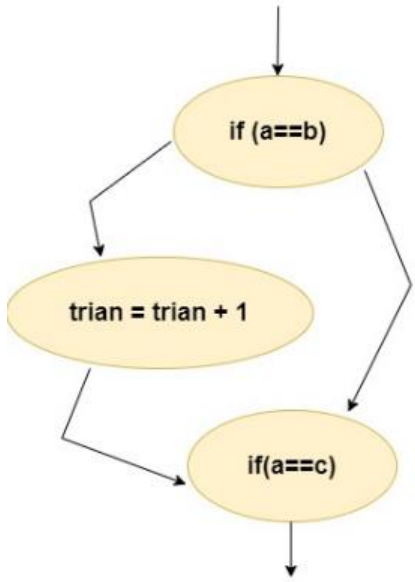


Figure 3(b).The CFG of lines 7 to 9

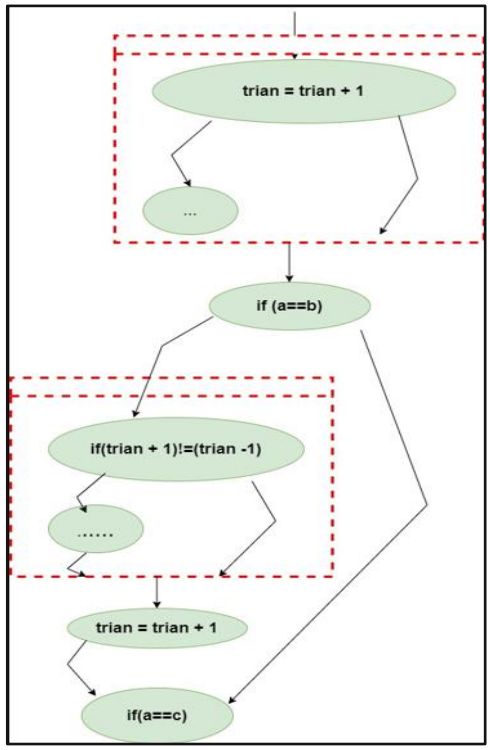


Figure 3(c). CFG after inserting mutant branches

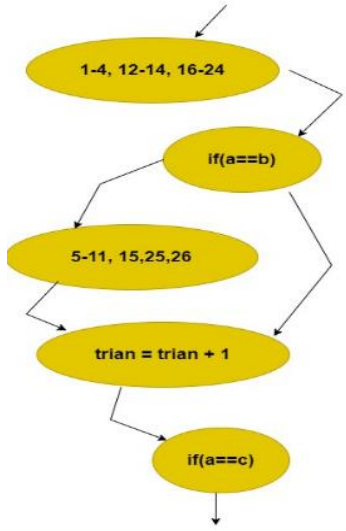


Figure 3(d). CFG of new program

Establishing the Dominance Relationship among Mutant Branches

The divisions in P that comprise both the actual and the variant ones are not entirely autonomous, despite the reality that the fusing of variant divisions enhances structural complexity. For instance, if mutant branch 15 in Table 3 and Fig. 3(d) is implemented, 16 to18 must also be carried out. The explanation is as follows. First, the reachability requirement for the three branches mentioned above is the same. Second, whereas the conditional expressions of 16 and 18 may be reduced to "a_1 == b_1||a_1 b_1" and "a_1 == b_1||a_1> b_1," respectively, the contingent interpretation of 17 can be reduced to "a_1 b_1." Since "a_1 b_1" is a prerequisite for both "a_1 == b_1||a_1 b_1" and "a_1 == b_1||a_1> b_1," any experiment piece of data that executes the real offshoot of 17 must also execute the branches of 16 and 18." Mutant branches 6, 9, and 10 must also be carried out if mutant branch 5 (the actual section) is carried out. The aforementioned findings support the assertions that (1) there is a connection amid mutant branches in P and (2) this correlation may be used to detect redundant mutant branches. The dominance relation is the name given to the correlation shown above. Table 5 illustrates the overall mutant operator and their respective operators after the final mutation process.

Table 2. The mutation operators and the number of mutants

Mutation operator	Mutants
AOIS	**33**
AOIU	**3**
AORB	**12**
COI	**3**
LOI	**9**
ROR	**15**
others	**0**
Total	**78**

b) Influence of Mutants over Influence Factor

Improvement factor (IF) is defined as the quotient between the time where no techniques are used over the time where one or more cost reduction techniques are used**.** (IF) is estimated using the below equation.

$$IF = \frac{(M\}-(T_{complie} + T_{pust}) + T_{install} + \rho - T_{run}}{\pi(T_{complie} + T_{pust}) + T_{install}) + M\,\rho - T_{run}} \textbf{(1)}$$

Usually the (IF) factor stabilises when the count of variantsincreases. Given that the assumptions used in the theoretical models are reasonable, we can use any values for the various indicators in Eq. 1 to represent the trends. The proposed mutation model stabilises the (IF) factor by using minimum number of mutants.Tools version, description details shown in table 2a.

4. Experimental Analysis

Table 2a. Experimental Analysis

S.No	Tools	Version	Description
1	Xamarin	5.0	Developing mobile application
3	Matlab	R2021a	Developing applications
4	Java	15	Developing applications
5	Mogo DB	5.0	Database
6	Linux	5.4. 0-26	Operating System

Table 3. The predicates of the constructed mutant branches

Mutant Operator	**Predicate**	
AOIS	1. if ((c == d) != (++c == d)); 2. if (c == d) != (− − c == d));	3. if ((c == d) != (c == ++d)); 4. if((c == d) != (c == − − d));
	5. if ((ans + 1) != (++ans + 1)); 6. if ((ans + 1) != (− − ans + 1);	21. if ((c == d) != (c++ == d)); 22. if ((c == d) != (c − − == d));
	23. if((c1 == d1) != (c1 == d1++)); 24. if ((c1== d1) != (c1 == d1 − −));	25. if ((ans + 1) != (ans++ + 1)); 26. if ((ans + 1) != (ans − − + 1)).
AOIU	7. if ((ans + 1) != (-ans + 1)).	
AORB	8. if ((train + 1) != (ans - 1)); 9. if ((ans + 1) != (ans∗ 1));	10. if ((ans + 1) != (ans / 1)); 11. if ((ans + 1) != (ans % 1)).
COI	12. if ((c_1== d_1) != (!(c_1 == d_1))).	
LOI	13. if ((c_1 == d_1) != (~ c_1 == d_1)); 14. if ((c_1 == d_1) != (c_1 == ~ d_1));	15. if ((ans + 1) != (~ans + 1)).
ROR	16. if ((c_1 == d_1) != (c_1< d_1));	17. if ((c_1 == d_1) != (c_1<= d_1));
	18. if ((c_1 == d_1) != (c_1 != d_1));	19. if ((c_1 == d_1) != (c_1>= d_1));
	20. if ((c_1 == d_1) != (c_1> d_1)).	

Figure 4 given below, describes the overall flow of the proposal. Initially the sample

application is taken as input. The application is synthesized as individual tokens. The next stage is muting the tokens. After muting the muted tokens are compared with the sample error tokens. If discrepancy is found, the actual application is free of errors else if discrepancy is found, the application has runtime bugs, which has to be further removed, to make the applications stable and free from runtime errors. The study uses Xamarin **4.8.0-sr2**, to develop the prime applications and implement the proposed framework. Different types of applications to synthesize the application code into individual tokens, muting the original token generated from the application and applications used to estimate the discrepancy between the original muted version of the application and sample error testing application. We have opted Xamarin tool due to its versatile characteristics such as native user experience, single technology stack, shared application logic, cost effective, integrated testing and easy maintenance.

Herewith we are displaying the execution of the testing model.

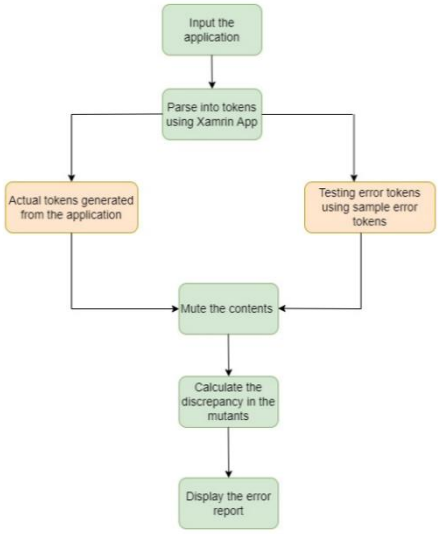


Figure 4. Flow of testing model

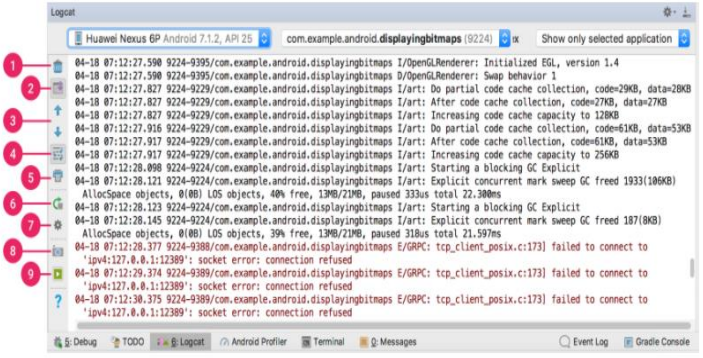


Figure 5. Discrepancy in muted and sample tokens

The above Figure 5 displays the discrepancy found in the muted and sample tokens of the application, which implies that the application is error free.

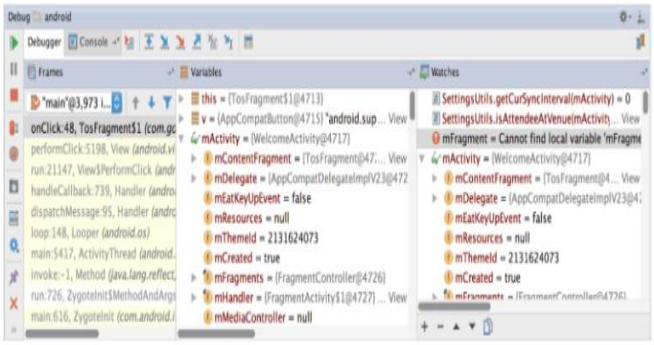


Figure 6. No discrepancy in muted and sample tokens

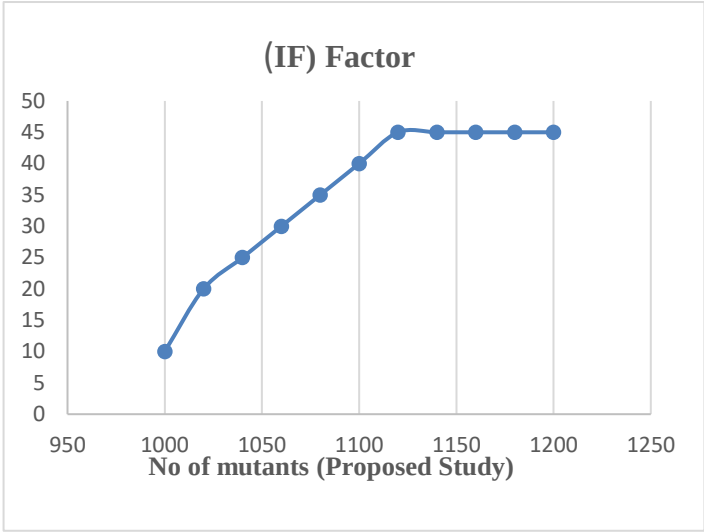
The findings shown above also show that there is no difference between the sample and muted error tokens, suggesting that the initial proposal may have some flaws that need to be fixed beyond to make it stable. The trials will use ten of the programs listed in Table 4. J1, J2, J3, and J4 are traditional mutation analyzing programs that first occurred in [2], [4], [30], and [31], respectively. J5 and J6 are sample programs from [32], and J5, J6, and J7 are research procedures from [36]. J8, J9, and J10 are programs for creating mutants [5]. All these are programmed in Java. For each program in Fig. 7, the transformation values increase instantly until they approach 80% (or even 90% in rare scenarios), and then steadily surge from 80% to the top. Just 12 test results are needed for J1 to achieve an 80 percent mutation score before mutant reduction. After reducing the number of mutants, only 9 samples are required to achieve the mutation result of 80%, while 22 samples are required to achieve the highest variant score (98.95%). However, 15 test results are needed from 80% to the highest percentage (98.25%). This implies that there are some mutants that are tough to kill and that it is hard to increase the transformation score much after it has reached a certain amount (such as 80 percent or 90 percent).

c) Comparison with other Related Works

(IF) factor among the mutants is estimated using our proposed mutant model and other related gene mutation model for testing android applications developed by [38]. Figure 8 clearly shows that the proposed model maintains the (IF) factor as the number of mutants increases. From the figure it is evident that in the proposed approach the (IF) factor stabilizes when the number of mutants is less over the referred study. This clearly indicates that techniques proposed work when they are actually needed. The proposed model checks (or) use it modules efficiently to test the needed Android applications. While in the referred study, their approach stabilizes the (IF) factor only when the number of mutants is increased more when comparing the proposed approach. The results are displayed in Figure (8).

Similarly, we have executed all the test cases against the mutants in eight variants for our proposed model.

We have executed 3 times the 31 test cases against the 1600 mutants using 1 and 2 devices. The experiments were performed under four categories namely (i)Mutant Schema (MS), (ii) No Mutant Schema (NoMS), (iii) All against all (AA) and (iv) Only Alive (OA). From the results obtained we find that there is negligible difference between the actual and estimated approaches, which prove the accuracy and efficiency of the proposed approaches. Table 4 illiterates the testing results. The suggested research involves a comparison between the empirical findings achieved for Android operators and the analogous methodology conducted in study [34]. Table 5 illustrates the outcomes, emphasizing that the proposed teasing model detects a greater number of mutants across all operators, subsequently eliminating them from the application, resulting in an enhancement of the mutation scores.



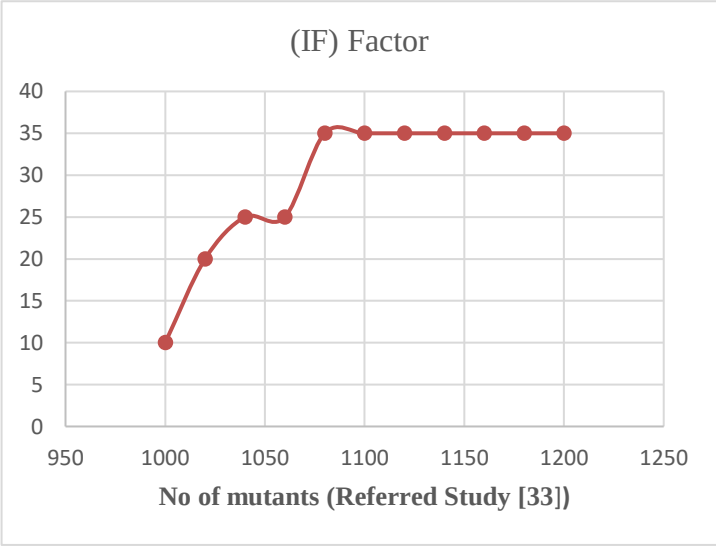


Figure 4. IF factor comparison

Table 4. Actual and estimated times in proposed approach

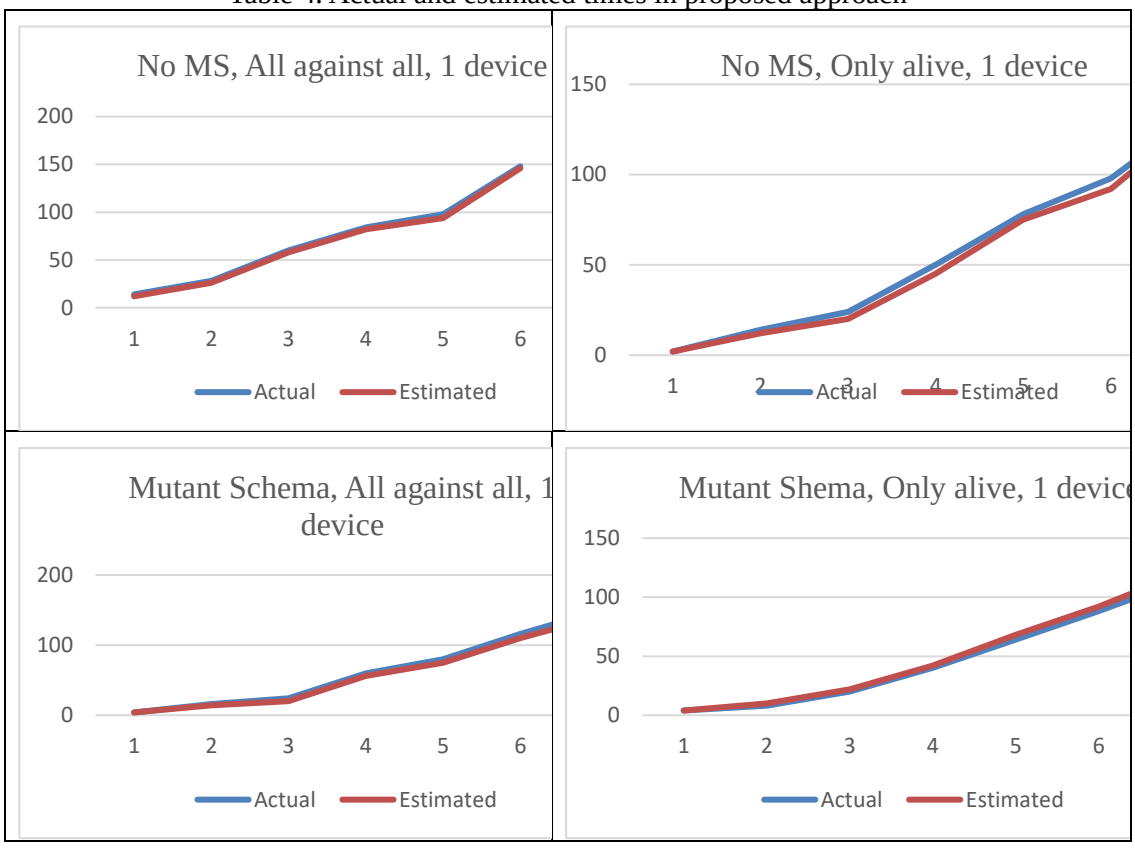


Table 5. Empirical results for android operators

Operator	**Refereed Study [34]**					**Proposed Study**				
	Killed Mutants	Equivalent Mutants	Live Mutants	Total Mutants	Mutation scores	Killed Mutants	Equivalent Mutants	Live Mutants	Total Mutants	Mutation scores
APD	10	4	21	35	0.321	25	4	10	39	0.5
IPR	7	0	0	7	1.000	17	0	0	17	1.000
ITR	181	0	29	290	0.862	240	0	0	240	0.982
ECR	111	0	4	115	0.965	125	0	1	126	0.978
ETR	2	0	0	2	1.000	12	0	0	12	1.000
FON	146	949	25	1120	0.854	181	23	2	206	0.938
MDL	18	1	5	24	0.783	22	1	0	23	0.865
BWD	36	0	0	36	1.000	54	0	0	54	1.000
TWD	6	0	4	10	0.600	14	0	0	14	0.728
ORL	13	0	35	48	0.271	25	0	10	35	0.581
BWS	0	0	99	99	0.000	0	0	12	12	0.858
Total	530	954	222	1786	7.656	715	28	35	778	9.43

5. Conclusion

This paper introduces a novel model that integrates advanced features of gene mutation techniques to effectively test mobile applications. The testing model proposed parses the mobile applications and checks for any existing runtime bugs which terminates the application without any intimation to the end user. Frequently used Android applications are considered for extensive testing. Different scenarios are used to identify the run time errors.The proposed model is developed using Xamarin mobile application development tool. Adequate APIs and interfaces for Xamarin applications are developed using Python. The experimental results prove that the proposed testing solution effectively parses and identifies the runtime errors effectively.

References

1. Gartner, Gartner says sales of smartphones grew 20 percent in third quarter of 2014. https://www.gartner.com/newsroom/id/2944819/
2. Google Play, (2015). https://play.google.com/store.
3. Aljedaani, W., Mkaouer, M. W., Ludi, S., Ouni, A., & Jenhani, I. (2022). On the identification of accessibility bug reports in open source systems. *In Proceedings of the 19th International Web for all Conference*, 1-11.
4. Johnson, J., Mahmud, J., Wendland, T., Moran, K., Rubin, J., & Fazzini, M. (2022, March). An empirical investigation into the reproduction of bug reports for android apps. *In IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 321-322.
5. Cui, D., Fan, L., Chen, S., Cai, Y., Zheng, Q., Liu, Y., & Liu, T. (2022). Towards characterizing bug fixes through dependency-level changes in apache java open source projects. *Science China Information Sciences*, *65*(7), 172101. https://doi.org/10.1007/s11432-020-3317-2.
6. Tan, S. H., & Li, Z. (2020). Collaborative bug finding for android apps. *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 1335-1347.
7. Li, Z., & Tan, S. H. (2020). Bugine: a bug report recommendation system for Android apps. *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 278-279.
8. Li, S., Guo, J., Fan, M., Lou, J. G., Zheng, Q., & Liu, T. (2020). Automated bug reproduction from user reviews for android applications. *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 51-60.
9. Cao, H., Meng, Y., Shi, J., Li, L., Liao, T., & Zhao, C. (2020). A survey on automatic bug fixing. *In 6th International Symposium on System and Software Reliability (ISSSR)*, 122-131.
10. Srinivasa Rao, M., Praveen Kumar, S., & Srinivasa Rao, K. (2023). Classification of Medical Plants Based on Hybridization of Machine Learning Algorithms. *Indian Journal of Information Sources and Services, 13*(2), 14–21.
11. Tian, Y., Yu, S., Fang, C., & Li, P. (2020). Furong: fusing report of automated android testing on multi-devices. *In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 49-52.
12. Adhikari, T. M., & Wu, Y. (2020). Classifying software vulnerabilities by using the bugs framework. *In 8th International Symposium on Digital Forensics and Security (ISDFS)*, 1-6.
13. Bhatia, A., Wang, S., Asaduzzaman, M., & Hassan, A. E. (2020). A study of bug management using the Stack Exchange question and answering platform. *IEEE Transactions on Software Engineering, 48*(2), 502-518.
14. Nakano, D., Yin, M., Sato, R., Hindle, A., Kamei, Y., & Ubayashi, N. (2020). A Quantitative

Study of Security Bug Fixes of GitHub Repositories. arXiv preprint arXiv:2012.08053.

15. Lee, D. G., & Seo, Y. S. (2020). Improving bug report triage performance using artificial intelligence based document generation model. *Human-centric Computing and Information Sciences*, *10*(1), 26. https://doi.org/10.1186/s13673-020-00229-7.
16. Pushpalatha, M. N., Mrunalini, M., & Sulav Raj, B. (2020). Predicting the priority of bug reports using classification algorithms. *Indian Journal of Computer Science and Engineering*, *11*(6), 811-818.
17. Jung, J., Kim, H. J., Cho, S. J., Han, S., & Suh, K. (2019). Efficient Android Malware Detection Using API Rank and Machine Learning. *Journal of Internet Services and Information Security*, *9*(1), 48-59.
18. Cide, Felip, José Urebe, and Andrés Revera."Exploring Monopulse Feed Antennas for Low Earth Orbit Satellite Communication: Design, Advantages, and Applications." National Journal of Antennas and Propagation 4.2 (2022): 20-27.
19. Mukherjee, D., & Ruhe, G. (2020). Analysis of compatibility in open source android mobile apps. *In IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, 70-78.
20. Mazuera-Rozo, A., Trubiani, C., Linares-Vásquez, M., & Bavota, G. (2020). Investigating types and survivability of performance bugs in mobile apps. *Empirical Software Engineering*, *25*, 1644-1686.
21. Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., & Liu, Y. (2020). An empirical study of usages, updates and risks of third-party libraries in java projects. *In IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 35-45.
22. Zhang, T., Hartmann, B., Kim, M., & Glassman, E. L. (2020). Enabling data-driven api design with community usage data: A need-finding study. *In Proceedings of the CHI Conference on Human Factors in Computing Systems*, 1-13.
23. Du, X., Zhou, Z., Yin, B., & Xiao, G. (2020). Cross-project bug type prediction based on transfer learning. *Software Quality Journal*, *28*(1), 39-57.
24. Wu, Q., He, Y., McCamant, S., & Lu, K. (2020). Precisely characterizing security impact in a flood of patches via symbolic rule comparison. *In the Annual Network and Distributed System Security Symposium (NDSS'20)*.
25. Chowdhary, M. S., Aishwarya, R., Abinay, A., & Harikrishna, P. (2020). Comparing machine-learning algorithms for anticipating the severity and non-severity of a surveyed bug. *In International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*, 504-509.
26. Park, M., You, G., Cho, S.J., Park, M., & Han, S. (2019). A Framework for Identifying Obfuscation Techniques applied to Android Apps using Machine Learning. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, *10*(4), 22-30
27. Jha, A. K., Lee, S., & Lee, W. J. (2019). Characterizing Android-specific crash bugs. *In IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 111-122.
28. Gu, Z., Wu, J., Liu, J., Zhou, M., & Gu, M. (2019). An empirical study on api-misuse bugs in open-source c programs. *In IEEE 43rd annual computer software and applications conference (COMPSAC)*, *1*, 11-20.
29. Activity Testing: What to Test, 2015, http://developer.android.com/tools/ testing/activity_testing.html#WhatToTest
30. Kong, P., Li, L., Gao, J., Bissyandé, T. F., & Klein, J. (2019). Mining android crash fixes in the absence of issue-and change-tracking systems. *In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 78-89.
31. Catolino, G., Palomba, F., Zaidman, A., & Ferrucci, F. (2019). Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, *152*,

165-181.
32. Petke, J., Harman, M., Langdon, W. B., & Weimer, W. (2014). Using genetic improvement and code transplants to specialise a C++ program to a problem class. *In Genetic Programming: 17th European Conference*, EuroGP 2014, 137-149.
33. Wu, F., Weimer, W., Harman, M., Jia, Y., & Krinke, J. (2015). Deep parameter optimisation. *In Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, 1375-1382.
34. JONNERBY, JAKOB, A. BREZGER, and H. WANG. "Machine learning based novel architecture implementation for image processing mechanism." International Journal of communication and computer Technologies 11.1 (2023): 1-9
35. Aydalga, S. C., Doğan, S., & Özkurt, A. (2020). Localisation and Museum Artifact Visual and Audio Presentation Using Bluetooth Beacon Technology. *Natural and Engineering Sciences,5*(2), 110-121.
36. Deng, L., Offutt, J., Ammann, P., Mirzaei, N. (2017).Mutation operators for testing Android apps. *Information and Software Technology*, *81*, 154-68.
37. Escobar-Velásquez, C., Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., & Poshyvanyk, D. (2020). Enabling mutant generation for open-and closed-source Android apps. *IEEE Transactions on Software Engineering*, *48*(1), 186-208.
38. Deng, L., Dehlinger, J., & Chakraborty, S. (2020). Teaching software testing with free and open source software. *In IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 412-418.
39. Chaparro, O., Bernal-Cárdenas, C., Lu, J., Moran, K., Marcus, A., Di Penta, M., & Ng, V. (2019). Assessing the quality of the steps to reproduce in bug reports. *In Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 86-96
40. Rodríguez-Trujillo ID. Mutation Testing Techniques for Mobile Applications.