MuDroid: Mutation Testing for Android Apps

Undergraduate Final Year Individual Project

Yuan Wei

BSc of Computer Science

Supervisor: Dr. Yue Jia

Abstract

With the raising of smartphones, mobile apps become an new emerging paradigm in software development. With 3.4 billion smartphone subscription, the quality of mobile apps become a crucial problem which lead to the need of novel and high quality testing approaches for app developers. Existing testing approaches, such as code coverage based, do not provide an effective way to assess the fault detectability of mobile app tests. The fault-based mutation testing techniques offer a better guideline in improving the fault detection ability.

This project aims to design and implement an automated mutation testing system for Android apps at integration level. With this system, Android developers and testers could gauge the quality of their test suites by understanding their fault detection ability. Furthermore, it can provide a guideline for testers to improve their testing as well as improving their implementation.

This thesis proposes three novel Android mutation testing techniques and describe the the implementation of MuDroid, an automated mutation testing system for Android apps. We designed six selective operators which generated 3,649 mutants on four real-world Android apps in the empirical study. The innovative screenshot-based killing approach we proposed could kill 40% more mutants on average than traditional crash-based killing approaches. The results also shown that our best cost reduction strategy could reduce on average 80% mutants with only a 7.3% error in average.

Contents

1 Introduction

According to the latest annual mobility report from Ericsson[1], there are about 3.4 billion smartphone subscriptions in the world. Moreover, each mobile app user could spend about 3 hours per day on the phone. These facts attract a large number of app developers. Until July 2015, there are about 3 million apps existed in Android and iOS market[2]. These apps generate about $23 billion revenue in 2015 from both Google Play and Apple app Store. Such a great market brings a more competitive environment for developers. Consequently, only apps with higher quality can survive in the market and generate revenues. Therefore, it is important for app developers to assure their apps work as expected. Otherwise, it will affect the experience of these millions of users and has a negative impact on the economy.

Mobile phone testing is vital for app development. In the real world, not all users use the app in the way developers expected. Therefore, it is important for app developers to test their app comprehensively. With the help of modern techniques, developers could implement an app in few weeks or even a few days. However, it is almost impossible for developers to find out all the bugs in the app in such a short time. As a result, users could come across many unexpected bugs or errors. To solve this problem itself brings a new question: What will be the criteria that should stop the testing and make the testing complete, and more importantly, how to find them? With the question raised, mutation testing is believed to be one of the answers.

1.1 Motivation

Mutation testing is a technique that seed artificial faults, known as mutants, into the code, then gauge the quality of tests based on the percentage of killed mutants. Comparing with traditional techniques like line coverage which only measures the percentage of executed code, mutation testing could identify the ability of fault detection for tests. It is capable of simulating the effect of other white box testing techniques and providing improved fault detection through automated fault injection. Thus, it offers a better standard in finding the completion criteria end the testing.

Although mutation testing has been proved to be an effective way to test programs, to date, little work has been done aiming for mobile app testing. Moreover, most existing mutation testing projects only focus on generic unit testing. For mobile app testing, developers are more interested in integration testing. Different from unit testing, integration testing for mobile apps targets the integration between each part of the entire app, instead of creating unit tests for the backend code. Particularly, it focuses more on the integration between the user interface and the backend code for Android apps. That could be a more challenging scenario as it requires a different form of testing and result collection. To date, to the best of our knowledge, there is no mutation testing framework available in the market for Android. Despite the fact that most Android apps were developed in Java, the integration testing of them requires different approaches. A novel approach for Android integration testing is needed to fill to gap.

1.2 Aim and Objectives

This project aims to design and implement an automated mutation testing system for Android apps at integration level. The implemented system could generate mutants for an APK file without access to the source code. In this mutation system, the quality of test suites is represented by the ability to kill mutants since each mutant could be treated as a small fault. Hence, with this system, Android developers and testers could gauge the quality of their test suites by understanding their fault detection ability. Furthermore, it can provide a guideline for testers to improve their testing. This project aims to achieve the following objectives.

- Define mutation operators for Android mutation testing.
- Design mutation selection strategies to improve the efficiency of mutation testing.
- Design a killing strategy for generated mutants.
- Implement an Android mutation testing system.

1.3 Contributions

We have made the following contributions in this project.

- A proposal of six selective operators designed for Android at Smali code level. In the empirical study, these operators generated 3649 mutants on four real-world Android apps.
- A proposal of novel screenshot-based killing approach. Result showed this approach could generally kill 40% more mutants on average than traditional killing approaches.
- A proposal of four selection strategies to improve the efficiency. In the evaluation, we observed the best strategy could reduce 80% mutants with a 7.3% error in average.

- An implementation of Android mutation testing system, MuDroid. MuDroid supports all the advanced mutation techniques for Android above. MuDroid has been used as a sensitivity analysis tool for Android apps in a collaboration project between Salerno University and UCL.

1.4 Report Outline

The rest of the report are organised as follow. Section 2 sets the background by describing the concept of mutation testing and Android testing. Existing works on Android mutation testing are also introduced. Section 3 outlines the details of designing the bytecode-based approach of Android mutation testing. Novel mutation operators, killings criteria and selection approaches are introduced accordingly. In Section 4, the design, implementation and testing for MuDroid, a mutation testing framework adopted the techniques proposed in Section 3, are explained in detail. Section 5 and 6 summarises the steps taken in evaluating the effectiveness and efficiency of the approaches described in Section 3. Limitations and future works were also discussed in this section. Finally, Section 7 summarises the key points of the work by briefly reviewing the contributions and achievements made throughout the project.

2 Background

This section introduces the background about this project. It first describes the fundamental hypotheses of mutation testing, the basic of mutation testing system and cost reduction techniques, followed by a discussion of Android mutation testing.

2.1 Mutation Testing

2.1.1 Fundamental Hypotheses

The idea of mutation was first proposed by DeMillo et al. in 1978[3] as a fault-based testing strategy. Its primary goal is to determine the ability of fault detection for given test suites. It has also been used to guide for test data generation [4, 5], as well as improve software non-functional properties [6, 7]. However, due to the tremendous number of possible faults of the program, it is necessary to choose a subset of these faults. This principle of subset selection is based on two theories, Competent Programmer Hypothesis (CPH) and Coupling Effect[3].

CPH states that programs developed by competent programmers are normally close enough to the correct version. In other words, most software faults written by experienced programmers are minor syntactic errors and could be corrected by small syntactical changes. Hence, only faults constructed from simple syntactical changes need to be selected.

The Coupling Effect states that "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [3]. It declares that complex errors are coupled with simple errors. Therefore, a test set is sensitive enough to detect complex faults if it could detect simple types of mistakes. According to these two theories, faults constructed from simple syntactical changes are sufficient for mutation testing. Test sets(suites) capable of detecting these simple faults should be good enough for complex fault detection.

2.1.2 Mutation System

Based the theories introduced above, mutation testing can be explained as a fault-based testing method which measures the fault detection ability of a given test set through quantifying the number of mutants being killed. Each mutant of program P contains small syntactical changes, and could be interpreted as a fault. Mutation testing system normally generates mutants {p', p"...} through mutation operators which introduce changes to the original statement. For example, line $b + c$ could be changed to different forms like $b - c$, $b * c$, b/c, $b\%c$, $b + (-c)$,

$(-b) + c$ and other forms, depending on the mutation operators. Each mutation operator normally contains a pattern to match and transformation rules to apply. In general, generated mutants will be sent to automated testing with a test suite. The test result of each mutant will be compared with the result from P. A mutant is marked as killed if the result of any test case in the test suite is different, otherwise, it is still alive. Figure 1 shows the process of mutation analysis.

Not all mutants generated will get killed eventually. Although program tester could provide additional test suites to increase the killed number of mutants, there are some mutants called Equivalent Mutants which could never get killed. They always produce the same result, even they are syntactically different. Thus, humans need to intervene in the equivalent mutants detection as it is impossible to detect these mutants automatically [8, 9]. Based on the ratio of the number of killed mutants K to the number of total mutants M subtracting the number of equivalent mutants E, the system gives a mutation score MS for program P and test set T which represents the ability of fault detection of the test set.

$$MS(P,T) = \frac{K}{(M - E)}$$

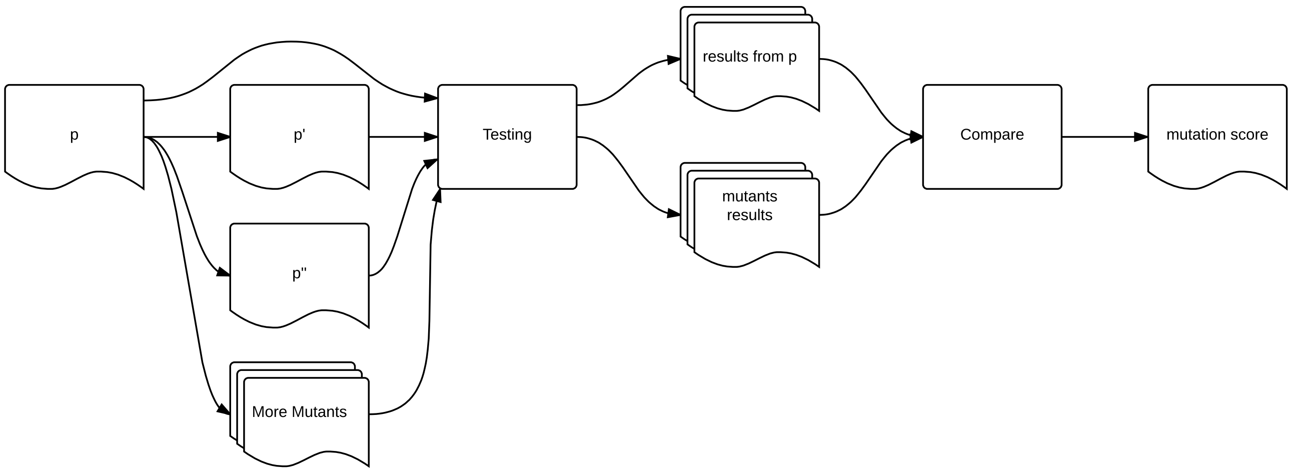


Figure 1: Mutation Analysis Process

2.1.3 Cost Reduction Technique

One of the major problems of mutation testing is the computational cost. Even a small-size program could generate a substantial amount of mutants. Such a large number of mutants could result in high costs in steps like compilation and execution. Thus, further steps need to be taken to reduce the costs.

To reduce the number of mutants get executed, Acree[10] and Budd[11] proposed the idea of Random Sampling. Instead of selecting the full set of mutants, Random Sampling only picks a small subset randomly. Further researches[12, 13] have shown that select 10 percent of the mutants over the full set is only 16% less effective, which is nearly adequate for a full mutation analysis. This finding is also agreed by other researchers[14, 15].

Instead of selecting mutants after generating all of them, another approach called Selective Mutation focuses on reducing the total number of mutants in the generation step. Selective Mutation tries to find a small set of mutation operators with which no significant loss of test effectiveness will be observed. According to the research of Offutt et al.[16], five key operators have been proved to be the most effective and could achieve 99.5 percent mutation score. These operators are ABS(Absolute Value Insertion), UOI(Unary Operator Insertion), LCR(Logical Connector Replacement), AOR(Arithmetic Operator Replacement), and ROR(Relational Operator Replacement).

Apart from selection, mutants can also be combined to reduce the overall cost, which is known as Higher Order Mutation Testing [17, 18]. First order mutants are created by applying a mutation operator only once, while higher order mutants are generated by applying mutation operators more than once. This work introduces a concept called subsuming higher order mutants, which is harder to kill than the first order mutants from which it is constructed. As a result, the total number of mutants can be reduced because first order mutants can be replaced by the fewer number of higher order mutants.

Another strategy used in cost reduction is reducing the execution cost. In traditional mutation testing, proposed by DeMillo et al. [3] in 1978, a mutant is marked as killed only if it gives a different output from the original program. That is, with the running of the entire program, the output obtained can then be used to determine whether a mutant is killed or not. To improve the efficiency, the concept of weak mutation was proposed by Howden[19]. Weak mutation brings a new model for the killing criterion of mutation testing. In this model, a program is assumed to be constructed by a set of components, and a mutant is killed if any of the results from these components is different from the original. As a result, instead of running the entire program to kill the mutants, mutants could be checked immediately after the execution of each component. This model brings the possibility of killing a mutant by unit tests. Since the original program, in general, should pass all test cases for unit testing, a mutant could be marked as killed if any of the test cases in the test suite failed.

2.2 Android Mutation Testing

Android is the operating system originally developed by Google for mobile phones. It takes APK(Android application package) formatted file for distributing and installing Android apps. APK file is an archive file which contains several files and folders like application's assets and compiled classes. It also contains the AndroidManifest.xml file which stores the information like application's package name and the start activity for the app.

Automated Testing is an important topic for Android development as it ensures the quality of the apps. Android Automated Testing mainly aims at either testing in unit or integration level. Unit testing is about testing a certain component of the entire system; while integration testing aims at the comprehensive system. Generally, a unit test targets at examining the functionally of the smallest piece of code possible. Local test and instrumented test are the two categories of unit tests in Android. Local test is almost identical to standard Java test and could get executed on local JVM(Java Virtual Machine) since it has no dependencies on Android framework. Unlike local test, instrument tests need to be executed on an actual Android device or an instance of the emulator as it requires Android framework components. However, none of these unit tests is suitable for UI interaction testing. Hence, integration testing is necessary.

Android integration testing involves the interaction between UI elements and functional code. The general approaches for automated testing in integration-level triggers test cases that consist of mobile app events like screen click, swipe, key input and other system events. These events can be generated and triggered randomly or based on GUI model. Google's Monkey[20] test tool is a random-based automated Android testing tool built into Android SDK. It generates and triggers Android events like screen tap, slide, inputs and physical button events. Other than generating events randomly, tools like AndroidRipper[21] and SwiftHand[22] could analyse the GUI model and generate user inputs based on the result. Other existing tools such as Calabash[23], Selendroid[24] and Appium[25] need pre-written scripts to run specific app events instead of generating them automatically.

Android apps are written in Java code and running on Dalvik[26], a specifically designed virtual machine for Android to replace the standard JVM. Not like PC, Android devices only have limited hardware resources like CPU speed and RAM. Therefore, Google decided to develop Dalvik, a tweaked version of JVM to improve the efficiency and to better suit the needs for mobile apps. For Android apps, source code written in Java will be compiled to Java bytecode first, and then translated to Dalvik bytecode and stored in *dex* files. Together with assets used in the app, *dex* files will be compressed to APK file for installation and distribution purposes.

Since Android application are mostly written in Java code, mutation testing tools like MuJava[27] and PIT[28] could be potentially used in the unit level testing. In order to improve the efficiency, these tools aim at Java Byte Code instead of Java source code. However, this approach also built a barrier for adopting these tools for Android mutation testing since Android applications are not executed on the standard Java virtual machine.

Lin Deng et al. introduced a technique for Android mutation testing in 2015[29] which brings eight novel operators and an innovative XML-based mutants generation approach. In their study, these operators generated 85 mutants and got a mutation score of 100% on an argumented test suite while traditional operators generate 190 mutants and only get 83.33% with the same test suite. Since this method targets at source code, it is not suitable for cases when source code is not accessible. Moreover, the developed mutation testing prototype tool for the experiment was not released to the public.

3 Mutation analysis for Android Apps

This section introduces the techniques we proposed for Android mutation system. It first describes the six selective mutation operators, followed by an explanation of screenshot killing criteria. Four novel selection strategies are also introduced along with a brief description of the traditional random sampling strategy.

3.1 Android Mutation Operators

As introduced earlier, mutation operators are the transformation rules used to generate mutants from the original code. Most Android applications are written in Java-like languages, the syntax is almost identical to Java, even though the source code will be compiled into the bytecode in a different format. Thus, most operators designed for Java could be adopted by Android as well. This project adopted LCR, AOR and ROR operators from the five operators model proposed by Offutt et al.[16] and implemented three new operators, ICR, NOI and RVR. ICR is a replacement of ABS operator as it is challenging to detect all numeric variables and to implement the absolute and failOnZero function at Dalvik bytecode level (Android bytecode format). Following this rationale, MuDroid uses ICR to change the value of constants assigned to variables as an alternative of ABS; UOI was also replaced by NOI, AOR, ICR and ROR due to the difference of bytecode and source code. In this six mutation operator model, most features of UOI was covered by ICR, AOR and ROR like inserting “++”, “-” and “!”” operators, while NOI works as a complement by inverting negative operators under special cases. Table 1 briefly describes the designed operators. These operators defined all the mapping and transformation rule for MuDroid.

Operator	Description
NOI	Invert negation of variables.
LCR	Replace logical connectors from one to the other.
AOR	Replace arithmetic operators from one to others.
RVR	Replace return value to *0* or *null*
ROR	Replace relational operator from one to others.
ICR	Change the value of constant before it get assigned to a variable.

Table 1: Selective Operators

The mutation of this project is based on Smali codes. Smali/baksmali[30] is an assembler/

disassembler for *dex* files which store the Dalvik bytecode. These Smali files contain bytecode instructions and the syntax for them is similar to assembler syntax. The whole process only targets bytecodes and does not require the original source code. Therefore, MuDroid could do mutation testing on any compiled Android APK files. However, as the project does not target on the source code level, it needs to define the operators at the bytecode level and use a different approach to implementing the transformations.

Smali has an assembler-like syntax; the normal instructions begin with a opcode followed with variables. As Dalvik is a register-based virtual machine, each variable normally represent a register. Labels are used to indicate positions in the code for jump-like instructions. Disassembled code within the same method in the original source code will be put into a block which starts with ".method" and ends with ".end". Similarly, instructions belong to the same line in the original source code will be put under a ".line" string with the line number. These bring the possibility for MuDroid to provide the location information of the mutants in the original source code.

3.1.1 NOI - Negative Operator Inversion

NOI was implemented to invert the value of variables. This operator was specially designed for Smali by detecting the occurrence of invert opcodes like "neg-x" and "not-x", where x represents the variable type like *int, float, double.* In Dalvik, these instructions could be used to handle "-" and "∼" operation. In Dalvik bytecode, there are various forms of statements for variable assignment. Since it is not practical to identify all variables and insert "-" and "∼" before them, the strategy used in MuDroid is just inverting all negative operators. Specifically, the program will duplicate the occurrence of "neg-x" or "not-x" operator once to get the corresponding inverted value.

Sourcecode	Bytecode	Mutation
x = ∼a	not-< *type* > vx, va	not-< *type* > vx, va not-< *type* > vx, va
x = -a	neg-< *type* > vx, va	neg-< *type* > vx, va neg-< *type* > vx, va

Table 2: NOI

```
gz va, :cond_1                          gz va, :cond_0
......                                  ......
gz vb, :cond_1                          gz vb, :cond_1
```

Figure 2: Pattern of && and ||

3.1.2 LCR - Logical Connector Replacement

LCR replaces each occurrence of logical connector(&&, ||) with each other connector. That is, replacing "&&" with "||" and changing "||" to "&&".

This operator looks entirely different in source code and bytecode level. Figure 3.1.2 illustrates the patterns of LCR, where *g* represents an operator from {*if-eq, if-ne, if-lt, if-ge, if-gt, if-le*} and *gz* is from {*if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez, g*}. The entire pattern for *g* and *gz* are very similar except *g* followed by two variables in the syntax of *g va, vc,* : $cond_1$ while *gz* followed by just one variable. The left side of Figure 3.1.2 shows the pattern of "a && b". If either *a* or *b* is false, it will jump to *cond_1* which normally either return false or points to the line after, if this logical connector occurs in statements like *for* and *if.* Right side is for the "||" connector. It points to *cond_0* when *a* is *true* and points to *cond_1* when *a* and *b* are both *false. Cond_0* normally leads to the section after the connector. If this connector occurs in an assignment, *cond_0* normally exists in the same line section and returns *true.* In addition, it points to the next line section if the connector is in checking sentence. Similar to the "&&" pattern, *cond_1* points to either return *false* or the next line section based on where the connector exists.

Table 3 shows the transformation for LCR operator, as states on the table, it needs to create an additional label *cond_0.* The label points to different sections of code based on the line where the connector exists.

3.1.3 AOR - Arithmetic Operator Replacement

AOR replaces each occurrence of arithmetic operators(+, -, *, /, %) with each other arithmetic operators. For example, $x = y + a$ will be transformed to $x = y - a$, $x = y * a$, $x = y / a$ and $x = y \% a$ where x and y are variables and a is a constant.

Most arithmetic operators have their corresponding instructions in Dalvik. However, $y - a$ is a bit different. For some types of variable, it does not have a direct subtract. Instead, it has a reverse subtract. Reverse instruction "rsub-int x, y, a" represents $x = a - y$ while the normal "sub-int x, y, a" represents $x = y - a$. Therefore, the sequence of variables needs to be

Sourcecode	Bytecode	Mutation
	if-eqz va, :cond_1	if-nez va, :cond_0
a && b	......	......
	if-eqz vb, :cond_1	if-eqz vb, :cond_1
	if-nez va, :cond_0	if-eqz va, :cond_1
a || b	......	......
	if-eqz vb, :cond_1	if-eqz vb, :cond_1
	if-gt va, vc, :cond_1	if-gt va, vc, :cond_0
$(a > c)\&\&(b > d)$	......	......
	if-gt vb, vd, :cond_1	if-gt vb, vd, :cond_1
	if-gt va, vc, :cond_0	if-gt va, vc, :cond_1
$(a > c)\|\|\|\|(b > d)$	......	......
	if-gt vb, vd, :cond_1	if-gt vb, vd, :cond_1

Table 3: LCR

reversed when handling *rsub* opcode. However, the syntax for this instruction requires c to be a constant making the reversing impossible. Another solution is inverting both y and a to get the result like x = *-a - (-)y* to represent $x = y - a$. However, this is also not possible as the Samli syntax does not allow "-" symbol exist before variables. Hence, an alternative solution is just using $x = -y + a$ for those types do not support *sub* opcode like 16bit and 8bit integers. Additionally, the syntax for 16bit int reverse subtract is a bit different with other opcodes. All other 16bit int arithmetic opcodes have the suffix "/lit16" like "add-int/lit16" while the *rsub* opcode for int16 is just "rsub-int". Table 4, Table 5 and Table 6 show the conversion for AOR operators.

Sourcecode	Bytecode	Mutation
x = y + a	add-int/lit8 vx, vy, a	any other arithmetic opcodes
x = -y + a	rsub-int/lit8 vx, vy, a	any other arithmetic opcodes
x = y * a	mul-int/lit8 vx, vy, a	any other arithmetic opcodes
x = y / a	div-int/lit8 vx, vy, a	any other arithmetic opcodes
x = y % a	rem-int/lit8 vx, vy, a	any other arithmetic opcodes

Table 4: AOR for 8bit integers

Sourcecode	Bytecode	Mutation
x = y + a	add-int/lit16 vx, y, a	any other arithmetic opcodes
x = -y + a	rsub-int vx, vy, a	any other arithmetic opcodes
x = y * a	mul–int/lit16 vx, y, a	any other arithmetic opcodes
x = y / a	div–int/lit16 vx, y, a	any other arithmetic opcodes
x = y % a	rem–int/lit16 vx, y, a	any other arithmetic opcodes

Table 5: AOR for 16bit integers

Sourcecode	Bytecode	Mutation
x = y + a	add-< *type* > vx, vy, a	any other arithmetic opcodes
x = y - a	sub-< *type* > vx, vy, a	any other arithmetic opcodes
x = y * a	mul-< *type* > vx, vy, a	any other arithmetic opcodes
x = y / a	div-< *type* > vx, vy, a	any other arithmetic opcodes
x = y % a	rem-< *type* > vx, vy, a	any other arithmetic opcodes

Table 6: AOR for all other types of variables

3.1.4 ROR - Relational Operator Replacement

AOR replaces each occurrence of relational operators($>, >=, ==, <, <=, ! =$) with each other relational operators. For example, $x > a$ will be transformed to $x >= a, x == a, x < a, x <= a$ and $x! = a$.

ROR implementation is rather straight-forward in Dalvik bytecode as each relational operator has a corresponding instruction. Table 7 shows the rule in detail.

Sourcecode	Bytecode	Mutation
a > b	if-gt va, vb	any other relational opcodes
a >= b	if-ge va, vb	any other relational opcodes
a == b	if-eq va, vb	any other relational opcodes
a <= b	if-le va, vb	any other relational opcodes
a < b	if-lt va, vb	any other relational opcodes
a ! = b	if-ne va, vb	any other relational opcodes

Table 7: ROR

3.1.5 RVR - Return Value Replacement

RVR replaces the value or object a function returned with *0* or *null*. It is a new operator specially designed for Android Dalvik bytecode. This operator widely exists in Android codes as Android code normally contains more functions due to the existence of event handler.

Sourcecode	Bytecode	Mutation
return i(variable in 32 bits)	return vA	return-void
return d(variable in 64 bits)	return-wide vA	return-void
return object	return-object vA	return-void

Table 8: RVR

3.1.6 ICR - Inline Constant Replacement

As mentioned above, ICR is a replacement of ABS. It replaces the value in the inline variable assignment. The operator exists in the form like *const< type > vx, 0xXX* where *type* refers to the variable type and *0xXX* is a hex value. In bytecode, this form contains not only ICR operator but also other values like the id of Android UI controls and other system constants. The basic idea of this operator is to change *1* to *0*, and all other values to the original value plus one so that system always get a different value. What's more, as boolean values are represented in integer values, this conversion also covers the boolean conversion from true to false and false to true. Table 9 shows this concept in the bytecode format.

Sourcecode	Bytecode	Mutation
x = 1	const< *type* > vx, 0x1	const< *type* > vx, 0x0
x = 0	const< *type* > vx, 0x0	const< *type* > vx, 0x1
x = other value	const< *type* > vx, 0xXX	const< *type* > vx, 0xXX+1

Table 9: ICR

3.2 Mutant Selection Strategy

The simplest and easiest strategy to choose mutants is applying all mutants generated without any filtering or selection. Since an APK file could generate numerous mutants from the designed mutation operators, the execution for these mutants could cost an enormous amount of time. Thus, it is relatively impractical to use this strategy in the real world due to its low efficiency. To improve the efficiency of MuDroid, we proposed four mutation selection strategies and compared them with the Random Sampling strategy which is a commonly used cost reduction strategy.

3.2.1 Random Sampling

Random sampling is an efficient way to reduce the number of mutants by randomly selecting x% from the total mutants. As the sampling strategy only takes a small subset of mutants, the result will not be as effective as the original: the effectiveness varies with different values of x. Wong and Mathurs studies[12, 13] showed that the mutation score is only 16% less effective than the full set when x is 10. In our evaluation, random sampling was used as a baseline to compare with the four proposed selection strategies below.

3.2.2 Inline Random Selection

Inline random selection selects one mutant among the mutants generated from the same line in an unbiased way. In particular, these are two different kinds of lines in MuDroid, Smali code line and source code line.

Smali Code

As mentioned above, compiled source code will be translated to Smali code in an assembly-like syntax. Each line normally could only be matched to one mutation operator but could generate more than one mutants from the operator. For instance, the line *add-int/lit8 va, vy, a* could generate *add-int/lit8 va, vy, a, rsub-int/lit8 va, -vy, -a, mul-int/lit8 va, vy, a, div-int/lit8 va, vy, a*, and *rem-int/lit8 va, vy, a.* This strategy will randomly pick one among these.

Source Code

Instead of choosing one mutant from the same line of Smali code, another approach is selecting one mutant among the mutants generated from the same line in original source code. In Smali code, one line of code could only generate the same type of mutants. However, in source code, one line may generate more than one type of mutants. For instance, *if(*$x > 2$ && $y < 1$*)* contains one LCR and two RORs. This selection will randomly choose one among them.

3.2.3 Equalization Selection

This selection method tries to equalise the amount of mutants by each mutation operator. Specifically, it selects mutants generated from the same source line by compensating the types with fewer mutants generated. As mentioned above, one line in source code could generate different types of mutants. Other than randomly pick one mutant among them, this method selects the type which currently has the lowest amount.

3.2.4 Pattern-based Selection

Pattern-based selection in another selection method for choosing mutants generated from the same mutation operator. Mutation operators like AOR and ROR generate more than one mutants; this method selects the mutations by certain patterns. The current pattern uses are boundary pattern and invert pattern. The boundary pattern is for ROR mutation operator only and the invert pattern is for both AOR and ROR. Boundary pattern selects mutants by adding or removing boundary conditions; invert pattern selects mutants by choosing its opposite. The details are shown in the tables below.

Original	Mutant
>	>=
>=	>
<=	<
<	<=

Table 10: ROR Boundary Pattern

Original	Mutant
==	! =
! =	==
>	<=
>=	<
<=	>
<	>=

Table 11: ROR Invert Pattern

Original	Mutant
$+$	$-$
$-$	$+$
$*$	$/$
$/$	$*$
$\%$	$*$

Table 12: AOR Invert Pattern

3.3 The Screenshot Killing Criterion

As mentioned earlier, normally, mutation testing tool takes input from the test program outputs. However, this strategy is not suitable for MuDroid, as it targets at integration testing instead of unit testing. Unlike unit testing, integration testing may not have a clear output at the system level. Thus, it is not possible to kill the mutants in the same form of unit testing. A new approach needs to be designed to obtain the result of Android integration testing. In that case, the most obvious way to obtain the output of testing result is observing changes in GUI.

In MuDroid, each mutant generated contains a small fault. With this fault, the app should behave differently. These differences will be reflected in the app's GUI or bring crashes. Crashes could also be detected by comparing the screenshots as Android will quit the app and produce a message box.

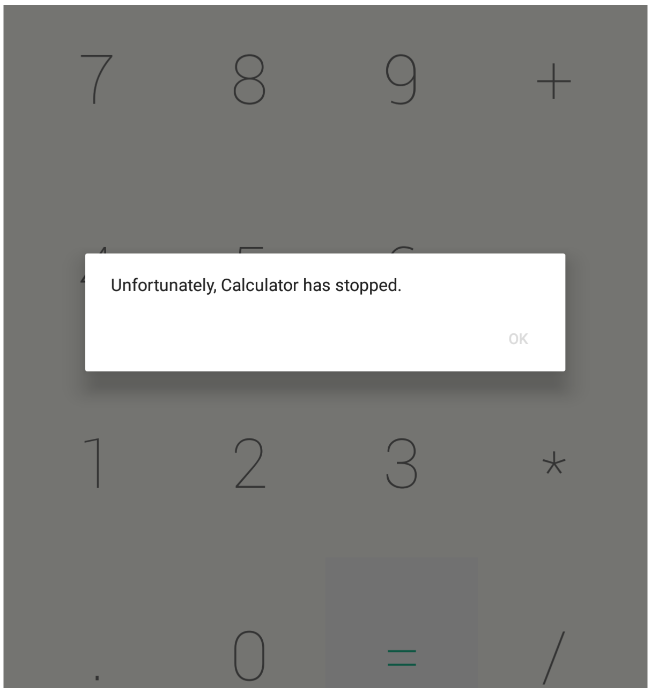

Unfortunately, Calculator has stopped.

Figure 3: Android App Crashes

Based on these, MuDroid uses screenshots from the app as its running result to kill mutants. It detects changes by comparing screenshots from APK mutants with the original APK. When executing mutants, screenshot will be taken for each Android event like click or swipe. These screenshots will be numbered by events' executing sequence. Only screenshots with the same number will be compared and will be treated different unless they are identical. Each APK mutant only contains one mutant and will be marked as killed if any screenshots is distinct from the original.

This strategy brings challenges as well. In order to improve the efficiency and get the most accurate result, screenshots need to be taken automatically after each action in the test while the app is running. In Android platform, each app is running in a sandbox and does allow other apps outside the box interacting it directly. Also, changes can also come from the system other that the app itself like time information.

To overcome these challenges, MuDroid uses *adb*[31] to interact the app instead of creating a background service app running on Android. As described in the official website: "Android Debug Bridge (adb) is a versatile command line tool that lets you communicate with an emulator instance or connected Android-powered device"[31]. With the aid of *adb*, MuDroid could send actions like tap or take screenshots directly to the Android device or emulator. As *adb* needs to send commands from PC to Android, the speed of performing these actions will not be as fast as running them natively. When taking screenshots, instruction need to be sent first from MuDroid through *adb*, then the device performed the action and stored the result on the device. MuDroid then needs to move this file from Android device to PC as the space on Android could be limited. These steps could be relatively slow based on the condition of available hardware resources. The accuracy of the result and the efficiency of the system are mutually exclusive in this condition. More screenshots make the result more accurate as it tracks more results but make the mutation checking more expensive. When comparing images, another issue needs to deal with is the minor change from the system like cursor blink or time change. Process these changes might make the comparison more expensive. For the information change like time or battery level, as they all appears on the status bar at the top of the screen, the image need to clipped to exclude the status bar. For the minor changes like cursor blink shown on Figure 5, thresholds are set for differences in pixels in x- and y-axes. The image will be treated as identical unless differences in both axes exceed the threshold.

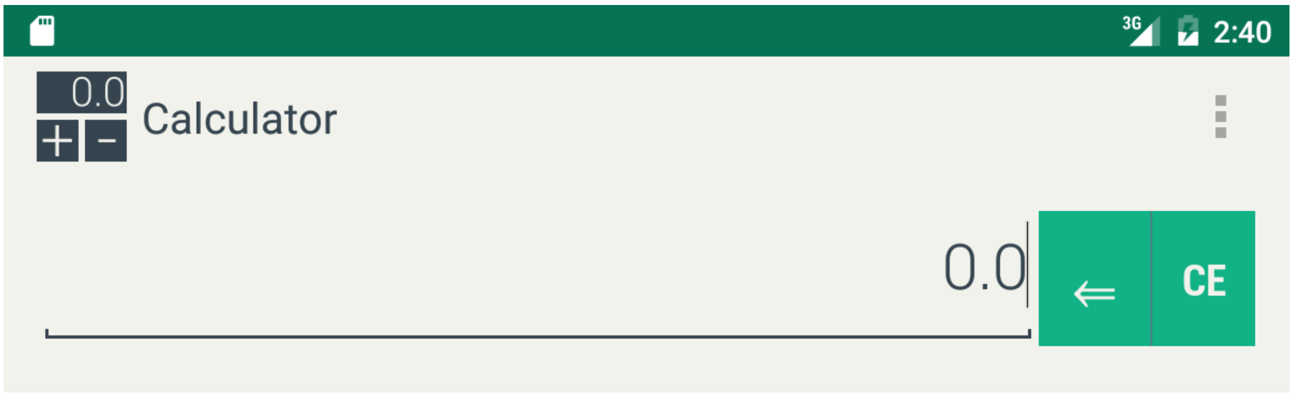


Figure 4: Status Bar

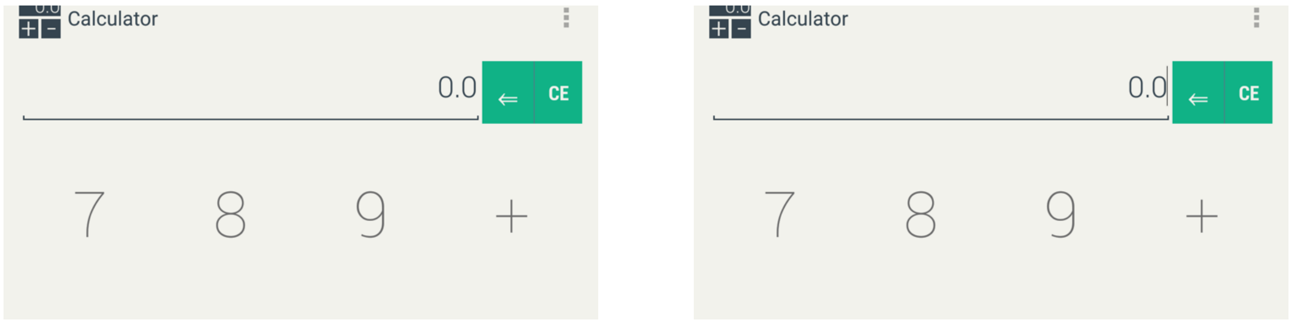


Figure 5: Cursor Blink

Although this killing criterion works for most mutants, it is will not be able to detect back-end changes that won't affect the front-end GUI or bring crashes. However, MuDroid is designed to measure the quality of interaction testing in the integration level. It is not possible to detect these changes automatically by the system in interaction testing. However, they could be easily distinguished by the developer of this app. Therefore, mutants contain back-end changes which will never bring a UI change will be reported to developers and not be further considered.

4 The MuDroid Tool

This section introduces the design and implementation of MuDroid, a system level mutation testing tool for Android apps. MuDroid features the Android mutation operators, the new selection and optimization strategies proposed in Section 3. This section also shows the unit and integration testing for MuDroid.

4.1 Design and Implementation

4.1.1 Overall Architecture

Figure 6 shows the overall architecture of MuDroid. MuDroid has a pipeline like structure which mainly contains three parts: a mutant generator, an interaction simulator and a result analyser. Each part could run independently without affecting others and has clearly specified inputs and outputs. This design allows users to use MuDroid in generating mutants, simulating user behaviours and analysing test result without executing the entire system.

MuDroid generates a report contains mutation score and mutants information for a given APK. In general, the input APK file will be sent to Mutant Generator to generate APK mutants. Interaction selection then takes these mutants and triggers UI tests which produces screenshot output. These screenshots will be sent to Result Analyzer to get the final report which shows the detail of these mutants and the mutation score.

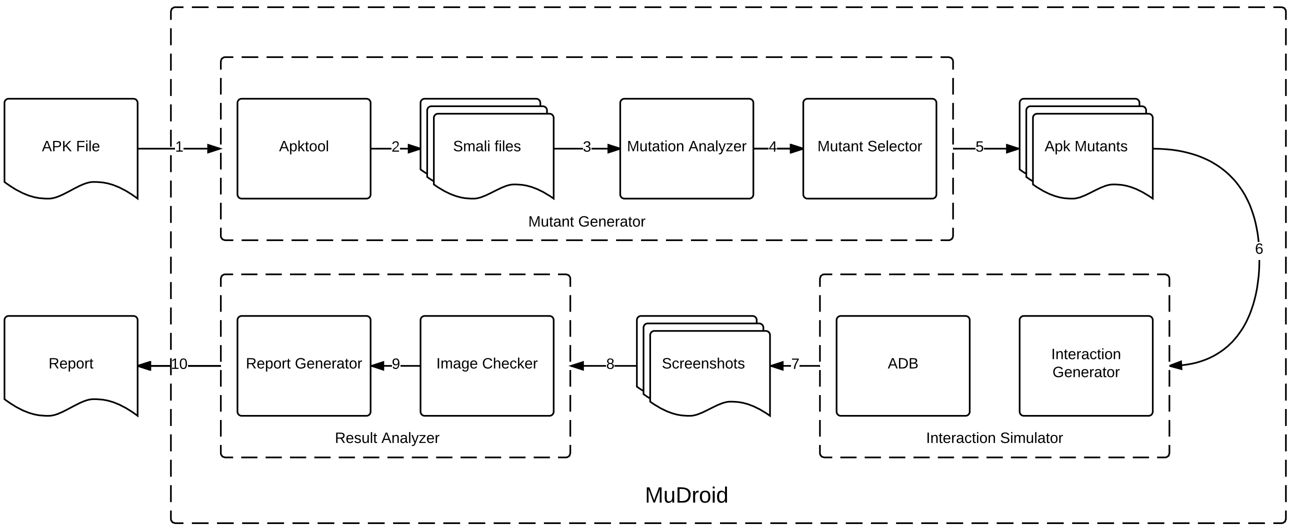


Figure 6: MuDroid System Architecture Diagram

4.1.2 Mutant Generator

Mutant Generator is the first components of the MuDroid pipeline. Its main job is to generate APK mutants from the APK file under test. Each APK mutant is an APK file containing one unique seeded fault. These mutants will be sent to Interaction Simulator for further processes. Mutant Generator consists three components: the *apktool* library, a mutation analyser, and a mutant selector.

apktool

apktool[32] is a tool could be used to compress/decompress APK files and decode/encode *dex* files. As mentioned, the decompressed APK file contains compiled classed in the *dex* file format which could be understood by Dalvik virtual machine. *apktool* could decode a *dex* file to Smali files or encode Smali files back to a *dex* file. Generally, *apktool* does the decompress and decode or compress and encode automatically in one step.

Mutation Analyzer

The Mutation Analyzer is one of the most significant components of MuDroid. Its job is to scan and generate mutants based on the mutation operators defined. It takes Smali files as input and scans these files to match the patterns defined in mutation operators. Once a pattern is found, MuDroid then generates mutants based on the rules defined in this operator.

Mutant Selector

MuDroid could generate thousands of mutants from one APK file. Thus, it is necessary to reduce the cost of processing these mutants. As mentioned above, mutant selection strategy could reduce the number of mutants need to process thus improve the efficiency. As a result, Mutant Selector is designed to implement the selection strategies proposed in Section 3. Each strategy is encapsulated into a standalone method. Thus, it is possible to apply multiple strategies to the same set of mutants.

4.1.3 Interaction Simulator

Interaction Simulator is the second part of MuDroid pipeline. It takes the APK mutants generated by Mutant Generator as input and produces screenshots as output. It is designed to simulate user behaviour when using the app by triggering auto or manual generated events through *adb*. The result of these events is recorded in screenshots which were captured with a pre-defined interval.

adb

adb(Android Debug Bridge) is an official command line tool for developers to interact with Android devices or emulators. It offers the ability like installing/uninstalling APK, copying files from or to the device instance or issuing shell commands[33]. In MuDroid, *adb* is used to issue screen click events, capture screenshot and move the screenshots from device instance to PC.

Interaction Generator

The Interaction Generator is an optional component of Interaction Simulator. It only runs when no tests were provided as input. It applies Monte Carlo technique to generate UI events like click, slide and text input. It can be replaced by any Android test data generation techniques like EvoDroid[34].

4.1.4 Result Analyzer

Result Analyzer is the last part of MuDroid pipeline. It processes the screenshots captured during the previous step to check whether a mutant is killed. Precisely, a mutant is marked as killed if any screenshot captured during the testing is different from the corresponding image in the original. It also generates a report which shows the detail of each mutant for users to investigate on equivalent mutants.

Image Checker

Image Checker is designed implement the killing criteria defined above by checking whether two screenshots are similar or not. However, there may be a minor delay when *adb* capture screenshots. The delay is due to the current performance of the Android device instance which it not fully controllable. Furthermore, Android system has the transaction animation when events triggered which may dim part of the screen. As a result, the screenshots captured may contain that slight colour change which needs to be corrected as it may not actually contain a difference. To solve the issue, Image Checker turns the input images to grey-scale and then uses two distinct algorithms to detect the similarity of images. It firstly finds the minimum box that contains all the differences and then calculates the RMSD(root mean square deviation) for given images. RMSD is calculated in the formula below.

$$RMSD = \sqrt{\frac{\sum_{t=1}^{n}(x_{1,t} - x_{2,t})^2}{n}}$$

Report Generator

Report Generator generates HTML formatted report which shows the detail of the mutation result in a table. Table 13 explains the details of each column in the report table.

Id	The unique id used to specify a particular mutant. It could be used to trace the detail of mutant generation and simulation through the log. Mutant id can also be used in generating and executing a particular mutant alone.
Mutation Operator	Type of mutation operator used to generate the mutant.
Smali File	Name and path of the Samli file contains the mutant.
Line number (Smali)	Line number of the Smali line contains the mutant.
Line number (source code)	Line number of the corresponding source code line contains the mutant.
Method	Name of the method in the source code. This method contains the line which generates the mutant.
Original line	Content of the Smali line before mutation.
Mutated line	Content of the Smali line after mutation.
Killed	Indicate whether the mutant get killed or not during the test.

Table 13: HTML Report Format

4.2 Testing

Both unit and integration testing have been applied to test MuDroid. The testing for MuDroid mainly focuses on checking if the mutants are generated in the designed way and the integration of MuDroid and the Android device.

4.2.1 Unit Test

Table 4.2.1-22 are some example of the unit level functional tests designed to ensure that the mutants generation works as expected.

Description	Generate mutants with ICR operator for float value which stored in IEEE standard hex.
Input	const/high16 v2, 0x40e00000 # 7.0f
Expected	const/high16 v2, 0x41000000 # 8.0f

Table 14: Test Case 1

Description	Generate mutants with ICR operator for 16 bit integer.
Input	const/16 v0, 0x8
Expected	const/16 v0, 0x9

Table 15: Test Case 2

Description	Generate mutants with ICR operator for 4 bit integer in overflow boundary case. (The range 4 bit integer could store are -8 to 7, add 1 to 7 will cause an overflow. Thus MuDroid needs to handle the overflow to avoid failure)
Input	const/4 v0, 0x7
Expected	const/4 v0, -0x8

Table 16: Test Case 3

Description	Generate mutants with ICR operator for *high16* double. High 16 variable type in Smali only accepts the highest 16 bits from MSB(Most Significant Bit), thus the convert result needs to be changed to normal type(without *high16* keyword) if it uses more that 16 bit.
Input	const-wide/high16 v4, 0x4059000000000000L # 100.0'
Expected	const-wide v4, 0x4059400000000000L # 101.0

Table 17: Test Case 4

Description	Generate mutants with ICR operator for double when the value is 0. Also, test normal *high16* mutation as the hex value for 1 and 0 both only use the highest 16 bits.
Input	const-wide/high16 v4, 0x0000000000000000L # 0.0'
Expected	const-wide/high16 v4, 0x3ff0000000000000L # 1.0

Table 18: Test Case 5

Description	Generate mutants with ICR operator for normal double value. The result should be 1 bigger than the original.
Input	const-wide/high16 v4, 0x4059000000000000L # 100.0'
Expected	const-wide v4, 0x4059400000000000L # 101.0

Table 19: Test Case 6

Description	Generate mutants with AOR operator in normal cases. MuDroid should generate 4 mutants which all have different arithmetic operators.
Input	add-int/lit8 v8, v8, -0x2
Expected	mul-int/lit8 v8, v8, -0x2 div-int/lit8 v8, v8, -0x2 rsub-int/lit8 v8, v8, -0x2 rem-int/lit8 v8, v8, -0x2

Table 20: Test Case 7

Description	Generate mutants with AOR operator for rsub. MuDroid should generate 4 mutants and add lit16 keyword to each mutant.
Input	rsub-int v8, v8, -0x2
Expected	mul-int/lit16 v8, v8, -0x2 div-int/lit16 v8, v8, -0x2 add-int/lit16 v8, v8, -0x2 rem-int/lit16 v8, v8, -0x2

Table 21: Test Case 8

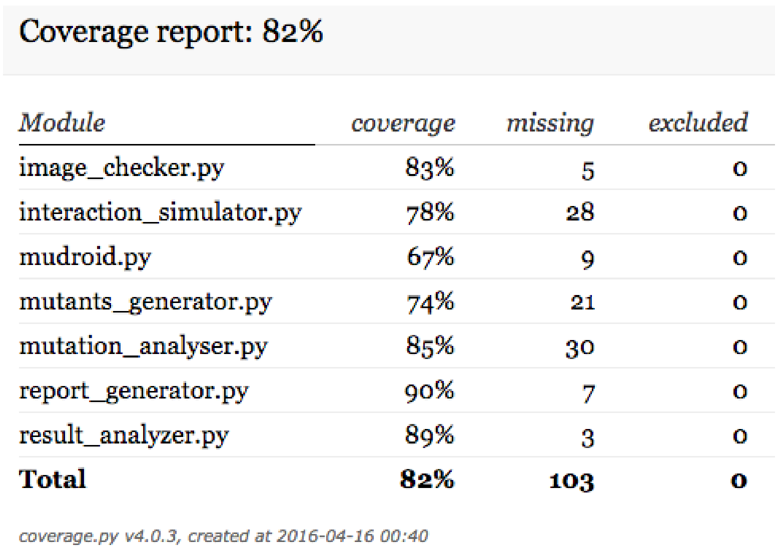
Description	Generate mutants with AOR operator for a non-rsub lit16 type variable. MuDroid should generate 4 mutants which all have lit16 keyword except rsub.
Input	add-int/lit16 v8, v8, -0x2
Expected	mul-int/lit16 v8, v8, -0x2 div-int/lit16 v8, v8, -0x2 rem-int/lit16 v8, v8, -0x2 rsub-int v8, v8, -0x2

Table 22: Test Case 9

4.2.2 Integration Test

As most MuDroid features require complex input, integration testing is also applied to test the system. A special testing app called MuDroidTester was developed to test the main features for MuDroid. MuDroidTester is a simple clicker app which shows a value when a button was pressed. Since the displayed value was calculated through few steps, mutants could affect the final result. Moreover, MuDroidTester also contains a few codes could affect the calculating result but will never get executed under normal condition. In other words, the result of these codes will only be seen with certain mutants. Thus, MuDroid needs to capture these change and kill the mutant during the execution of these mutants.

Figure 7 shows the code coverage of MuDroid when doing mutation testing with MuDroidTester. As it shown, 83% code of MuDroid were executed during the testing. Therefore, the test covers most part of MuDroid. The results were checked manually to ensure the functions provided by MuDroid are correct.



Coverage report: 82%

Module	*coverage*	*missing*	*excluded*
image_checker.py	83%	5	0
interaction_simulator.py	78%	28	0
mudroid.py	67%	9	0
mutants_generator.py	74%	21	0
mutation_analyser.py	85%	30	0
report_generator.py	90%	7	0
result_analyzer.py	89%	3	0
Total	**82%**	**103**	**0**

coverage.py v4.0.3, created at 2016-04-16 00:40

Figure 7: Code Coverage for Mutation Tester App

All defined operators could generate mutants with the MuDroid Tester App. During the test, 50 mutants were generated, every mutant was checked manually to make sure the it was produced in the correct way and contains accurate information. Source codes were also involved in the testing to ensure all possible mutants could be identified in source code level were generated. In addition, all screenshot captured were checked manually to ensure mutants were killed correctly.

5 Empirical Evaluation

This section first discusses the research questions we addressed in our empirical evaluation of the MuDroid tool, followed by an explanation of the chosen subjects and experiment settings.

5.1 Research Question

The following research questions were designed to investigate the effectiveness and efficiency of the Smali-based Android mutation testing approach.

RQ1 [Prevalence] What are the numbers and proportions of Android mutants found overall and per program studied?

The question was designed to discover the prevalence of the mutation operators defined. To answer this research question, the number of mutants generated by these operators will be analysed.

RQ2 [Effectiveness] How effective is Android mutation testing using MuDroid?

This question is about to discover the efficacy and effectiveness of the Smali-based Android mutation testing approach designed. We designed to set of test manually; one is in higher quality with more test cases, while the other has lower quality with fewer test cases. To answer this question, we run MuDroid to check whether it produces higher mutation score on the high-quality tests.

RQ3 [Killing Criteria] How effective is the screenshot-based killing criteria?

The question was designed to discover the effectiveness of the novel screenshots-based killing criteria. To answer this research question, we compare the number of mutants killed by crashes with the number of mutants killed by the screenshot killing criteria.

RQ4 [Selection Strategy] How efficient are the selection strategies?

MuDroid supports four mutants selection strategies along with the traditional 10% random sampling selection strategy. This research question investigates on the reduction of mutants and error in mutation score.

5.2 Experiment Subject

Four Android Apps from the Android market were selected for the empirical study. Table 23 shows the details for each app like the total line of source code and Smali code. All these Apps are available in either F-Droid[35] or the official GitHub repository.

Subject	Version	Line of Source Code	Line of Smali Code
agram	1.3	959	6,289
Clean Calculator	1.1	915	4,551
Did I ?	1.0.1	1,973	7,456
Droid Draw	1.0.0	3,568	21,371

Table 23: Experiment Subjects

These four subjects are in different categories and have different sizes in the line of code. *agram*[36] is an app could list single-word and multi-word anagrams in English for given word. According to Google Play, the most recent version 1.3 was released on 4 October 2015. The APK used for evaluation was downloaded from its F-Droid page[37].

Clean Calculator[38] is a scientific calculator app. This calculator has advanced mathematical functions like sinus, cosinus, in addition to the standard mathematical functions like addition and subtraction. It is a popular app in Android market as it has more than 10,000 installs. The APK used for this evaluation was released on 30 March 2015 and was downloaded from the F-Droid page[39]. The calculation relies on a third party library there MuDroid mutant that library instead of the main package of *Clean Calculator* to generate more calculation-related mutants.

Did I ? is a habit tracking app which asks users the question in the "Did I ... ?" form every day. Users need to answer these questions every day and the app will generate reports for users to track the progress completeness of these habits. For instance, if a user would like to do exercise every day, he could create a question like "Did I exercise?". He then needs to choose "yes" or "no" every day when the app asks this question. Based on the selections he made, the app tracks whether he did exercises or not every day. According to its F-Droid page[40], which is where the APK was downloaded, the version 1.0 was added on 5 June 2013.

Droid Draw[41] is an automated drawing app allows users to create complex shapes using part of the LOGO programming language. The APK was downloaded from the F-Droid page[42] and was released on 08 September 2012.

5.3 Settings

All the experiments were undertaken on the Mac Pro with 3.33GHz 6 core Intel Xeon processor and 16GB memory. All mutants were executed on Nexus 7 tablet or equivalent emulator. Table 24 shows the hardware and software detail for these two machines. MuDroid requires Android SDK installed and tested with *adb* version 1.0.32 and *apktool* 2.0.3. The version of Python libraries MuDroid relies on was also shown in Table 24.

Model	Mac Pro
OS	OS X Yosemite 10.10.4
Processor	3.33GHz 6 core Intel Xeon
Memory	16GB
Python Version	2.7.10
adb Version	1.0.32
apktool Version	2.0.3
Pillow Version	3.1.1
pexpect Version	4.0.1

Table 24: Experiment Setting

6 Results and Discussion

6.1 Answer to RQ1: Prevalence of the mutants

We begin by looking at the mutants generated for each subject by the designed mutation operators to answer RQ1. The overall result is reported in Table 25 and Figure 8. During the experiments, MuDroid generated a total number of 3,649 mutants with the six predefined mutation operators. The app *Droid Draw* generated about half mutants, one possible reason is that it has more lines of source code and Smali code than others. Figure 9 shows the relation of the line of code and mutants generated for each subject to investigate on the factor. We observed that generally more lines of code intended to generate more mutants as it has more input. However, it is not the factor or at least not the only factor since *Clean Calculator* generate 639 mutants with 4,551 Smali lines, which is more than the 572 mutants generated from the 7,456 Smali lines in *Did I ?* app. The ratio of source code lines, Smali lines and the number of mutants varies and does not follow a clear pattern. Thus, the number of mutants generated by these operators is not decided by the total number of Smali lines or source code lines alone.

Subject	ICR	NOI	LCR	AOR	ROR	RVR	Total
agram	111	0	10	28	360	73	582
Clean Calculator	58	0	8	48	470	55	639
Did I ?	107	0	0	132	295	38	572
Droid Draw	540	8	5	364	860	79	1,856
Overall	816	8	23	527	1,985	245	3,649

Table 25: Mutant Generation (Overall)

Among all the defined operators, ROR generates most mutants which take about 60.33% of the total mutants generated. Comparing with other operators, ROR and AOR operator could generate multiple mutants from the same line. To excluding this effect, an analysis was conducted to investigate on the mutants generated for each operator in the case that ROR and AOR only generate one mutant.

Table 26 and Figure 10 show the statistics of the mutants generate after clearing the case that AOR and ROR generate multiple mutants. RVR operator generated 29% of mutants, since this operator was designed for return operation, one RVR mutants normally represents

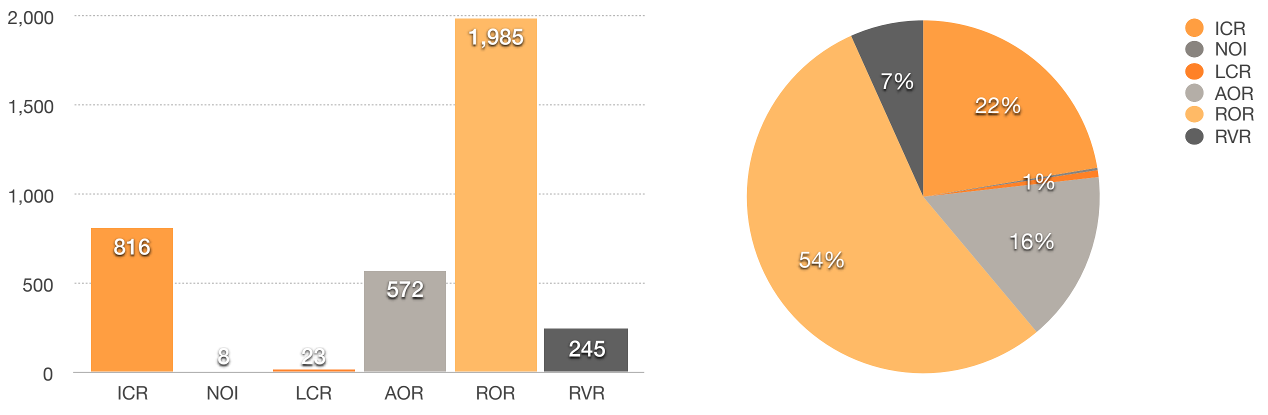
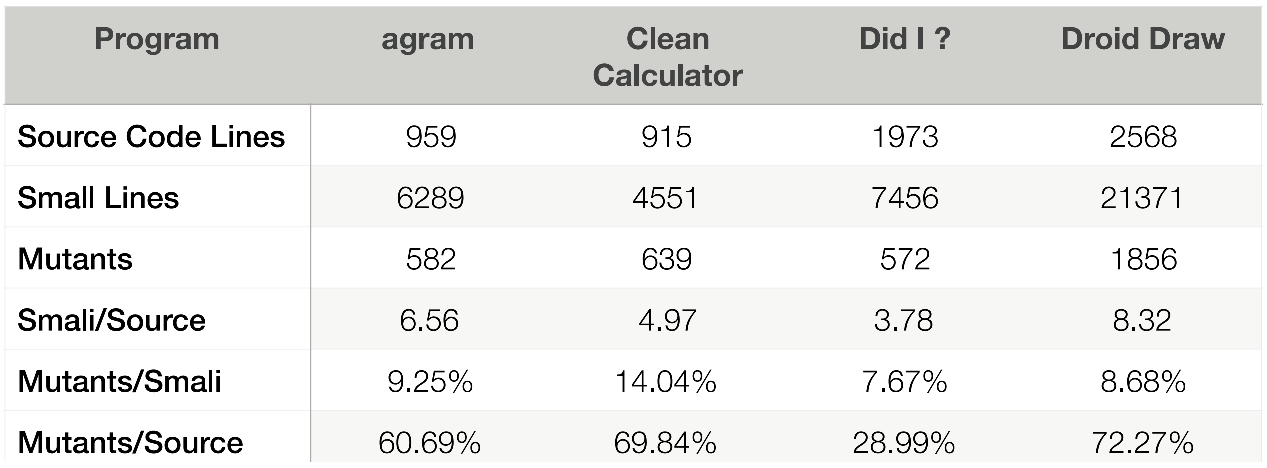


Figure 8: Percentage of mutants generated for each mutation operator(Overall)



Program	agram	Clean Calculator	Did I ?	Droid Draw
Source Code Lines	959	915	1973	2568
Small Lines	6289	4551	7456	21371
Mutants	582	639	572	1856
Smali/Source	6.56	4.97	3.78	8.32
Mutants/Smali	9.25%	14.04%	7.67%	8.68%
Mutants/Source	60.69%	69.84%	28.99%	72.27%

Figure 9: Relation between line of code and mutants generated

that there is a method in the source code. The existence of more RVR mutants means the app has more methods, in some cases, a higher ratio of methods to the total line of code means the app has a better structure.

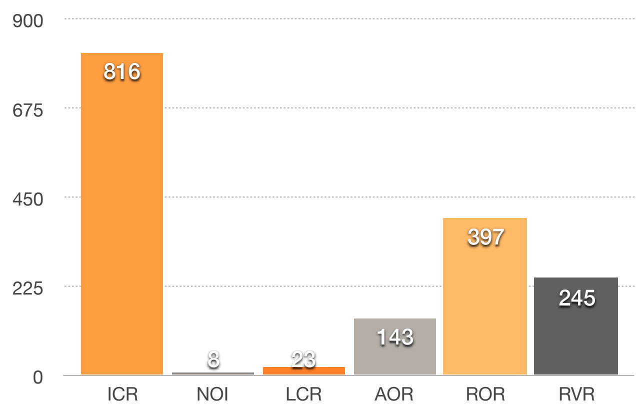
NOI operator was specially designed for Smali code as it has special instruction for unary operations to invert variables. However, we observed that it did not generate enough mutants as expected. Our investigation shows that the special instruction “neg-” only occurs when there is a direct inversion in source code line “$a = -a$” and will be replaced by arithmetical instructions in other cases like “$a = -a + b$”. It suggests the fact that Android developers may not intend to use the unary inversion often during the development.

6.2 Answer to RQ2: Effectiveness of MuDroid

In this experiment, two test suites T1 and T2 were manually created for each subject to evaluate the ability for MuDroid to detect the quality of test suites. T1 was designed to be a higher quality with more tests included, while T2 has a relatively low quality and containing a subset

Subject	ICR	NOI	LCR	AOR	ROR	RVR	Total
agram	111	0	10	7	72	73	273
Clean Calculator	58	0	8	12	94	55	227
Did I ?	107	0	0	33	59	38	237
Droid Draw	540	8	5	91	172	79	895
Overall	816	8	23	143	397	245	1,632

Table 26: Mutant Generation(Excluding multiple generation)



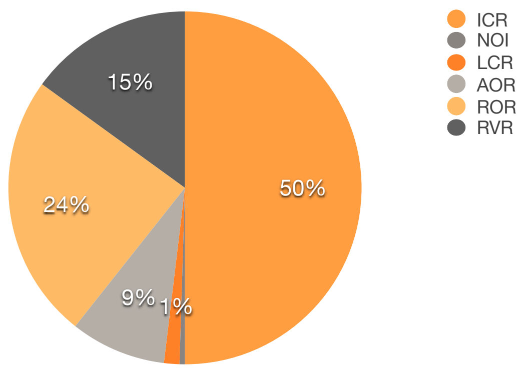


Figure 10: Percentage of mutants generated for each mutation operator(Excluding multiple generation)

of tests selected from T1. Each of these test suites contains test cases in the form of UI event like tap of text input. Generally, T2 contains 50% or fewer test cases than T1.

For this experiment, T1 hand T2 varies with different test subjects. *agram* has 19 test cases for T1 and 9 test cases for T2; *Clean Calculator* has 30 test cases for T1 and 15 test cases for T2; *Did I ?* has 41 test cases for T1 and 10 test cases for T2; *Droid Draw* has 32 test cases for T1 and 14 test cases for T2. These tests are running without any selection strategies to gather more accurate result. Therefore, to reduce the cost of execution, these experiments does not use a large set of test cases. As expected, the mutation testing score for T1 is always higher than T2 for all programs. This sanity check experiment shows that MuDroid is effective enough to be used as an Android mutation testing system.

6.3 Answer to RQ3: Effectiveness of Screenshot Killing Criterion

We now turn to the investigation of the efficacy of the screenshot killing criteria. We run MuDroid using both the traditional crash-based criterion and the new screenshot-based criterion

Subject	Mutation Score for T1	Mutation Score for T2	Diff
agram	0.6186	0.3574	0.2612
Clean Calculator	0.3991	0.3568	0.0423
Did I ?	0.5175	0.3741	0.1434
Droid Draw	0.2807	0.2705	0.0102

Table 27: Mutation Score for Each Program

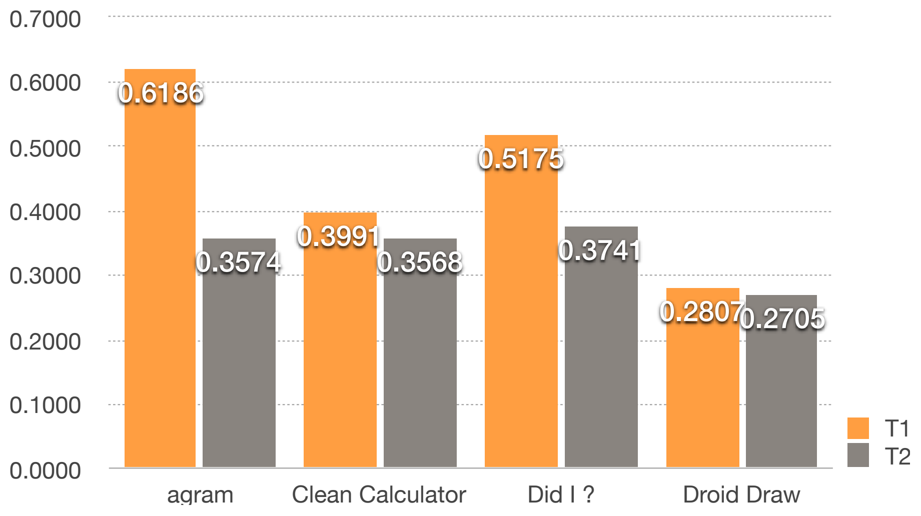


Figure 11: Mutation Score for T1 and T2

and report the results in Table 28. It is clear that the traditional killing criterion is not sufficient to kill all the mutants and the new killing criteria kill 34.83% to 400.00% additional mutants. We observed that the screenshot-based criterion killed four times more mutants for the *Clean Calculator* subject. The reason is that many mutants generated from *Clean Calculator* could affect the calculation process and the result will be reflected in the UI as it is a simple calculator app. Hence, for *Clean Calculator*, more mutants were killed by the screenshots comparison. Compare with traditional killing criteria like crashes detection, this image comparison approach is more efficient in killing mutants in Android integration testing.

Figure 12 shows the percentage of mutants killed by crashes and screenshots. According to this data, screenshot comparison could kill about 40% more mutants on average than only using crashes detection. Since *Clean Calculator* is more UI sensitive than other programs and killed far more mutants than others with the screenshots killing approach, a possible speculation is this approach may be more efficient in UI sensitive apps.

Subject	Mutants Killed(T1)	Killed by Crashes	Killed by Screen Comparison
agram	360	267	93
Clean Calculator	255	51	204
Did I ?	296	198	98
Droid Draw	521	375	146

Table 28: Mutants killed by crashes and screenshots

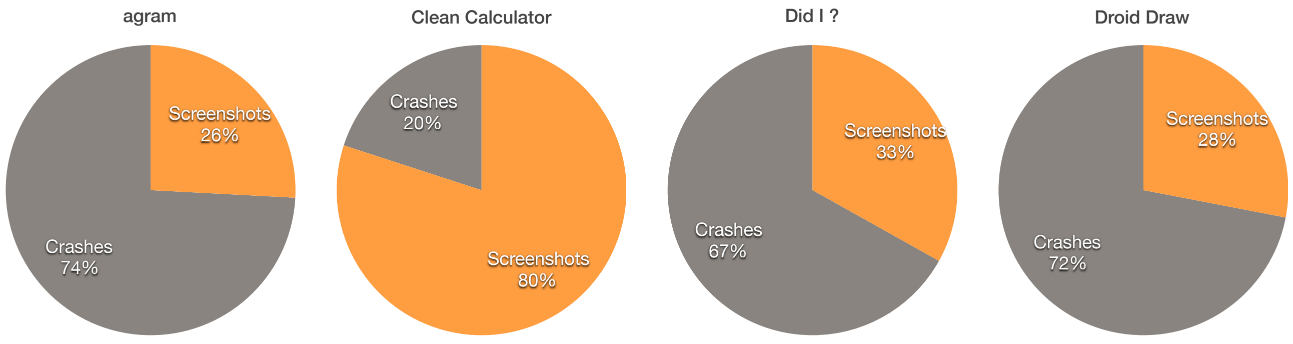


Figure 12: Percentage of mutants killed by crashes and screenshots

6.4 Answer to RQ4: Efficiency of Selection Strategies

Experiments were also undertaken to evaluate the selection strategies proposed above. We first run MuDroid without any selection as a baseline, follow by repeat the experiments using the selection strategies we proposed. Results from Random Sampling selection are used as the guideline in efficiency for others to compare with. Following tables shows the mutation score for each subject under each selection strategy, where *ms* represents the mutation score. Mutation score for random selection like random sampling and inline random selection is the mean value of 5000 times execution. All selection experiments were tested by T1.

Subject	Mutants Reduced	ms(Mean)	ms(Max)	ms(Min)	ms(Original)
agram	524	0.6193	0.8275	0.3621	0.6186
Clean Calculator	576	0.3991	0.6349	0.2063	0.3991
Did I ?	515	0.5149	0.7368	0.2982	0.5175
Droid Draw	1671	0.2806	0.4	0.1568	0.2807

Table 29: Random Sampling Selection

Subject	Mutants Reduced	ms(Mean)	ms(Max)	ms(Min)	ms(Original)
agram	383 (65.81%)	0.6655	0.7236	0.5980	0.6186
Clean Calculator	474 (74.18%)	0.4916	0.5515	0.4181	0.3991
Did I ?	359 (62.76%)	0.5790	0.6291	0.5305	0.5175
Droid Draw	1241 (66.86%)	0.2976	0.3317	0.2699	0.2807

Table 30: Smali Inline Random Selection

Subject	Mutants Reduced	ms(Mean)	ms(Max)	ms(Min)	ms(Original)
agram	467 (80.24%)	0.6778	0.7739	0.5739	0.6186
Clean Calculator	498 (77.93%)	0.4944	0.5674	0.4255	0.3991
Did I ?	433 (75.70%)	0.5404	0.6115	0.4676	0.5175
Droid Draw	1529 (82.38%)	0.2797	0.3272	0.2294	0.2807

Table 31: Source Code Inline Random Selection

Table 29, 30, 31 show the result of 5,000 times test for random selection strategies. Random sampling was tested with 10% mutants selected.

As these tables shown, 10% random sampling strategy could reduce most mutants, Smali and source inline random selection could reduce about 67% and 79% mutants. However, the result for random sampling could have an error in the range of -0.19 to 0.23 in terms of mutation score. In this case, source code inline random selection only the difference from -0.05 to 0.17 which is a better option.

Subject	Mutants Reduced	Mutation Score	Original Mutation Score
agram	467 (80.24%)	0.6522	0.6186
Clean Calculator	498 (77.93%)	0.5673	0.3991
Did I ?	433 (75.70%)	0.5612	0.5175
Droid Draw	1529 (82.38%)	0.3303	0.2807

Table 32: Equalization Selection

Apart from the unstable results random selection strategies, Table 32 and 33 show the result of equalization and pattern-based selection strategies. The result strategies intend to be stabled as these strategies do not bring any randomised factors. Equalization and pattern-based selection are the extensions of source code and Smali inline selection. Hence, they are compared

Subject	Mutants Reduced	Mutation Score	Original Mutation Score
agram	295 (50.69%)	0.6411	0.6186
Clean Calculator	397 (62.13%)	0.5207	0.3991
Did I ?	316 (55.24%)	0.5391	0.5175
Droid Draw	929 (50.05%)	0.2837	0.2807

Table 33: Pattern-based Selection

with their predecessors. Comparing with source code inline selection, equalization reduces the same amount of mutants and has smaller error range. Pattern-based selection only reduces 54.53% mutants in an average. However, the mutation score is more close to the original.

Table 34 shows the overall comparison of each selection technique. We observed that all strategies proposed do not reduce as many mutants as Random Sampling reduced but have smaller errors. We also notice that all proposed strategies intend to have a positive value of error. Equalization Selection and Pattern-based Selection do not even produce negative value errors and have smaller changing scope between the maximum and the minimum error. Although Equalization Selection has a slightly bigger error scope than Pattern-based Selection, it kills 24.53% more mutants on average. Consider of that; it is a better strategy than Pattern-based Selection. To summary, we suggest using Equalization Selection in MuDroid.

Strategy	Mutants Reduced	Error in Mutation Score
10% Random Sampling	90%	[-0.1928, 0.2358]
Smali Inline Random Selection	67.40%	[-0.0206, 0.1524]
Source Code Inline Random Selection	79.06%	[-0.0513, 0.1683]
Equalization Selection	79.06%	[0.0336, 0.1682]
Pattern-based Selection	54.53%	[0.0030 ,0.1216]

Table 34: Comparison of Selection Strategies

6.5 Limitation

Even though the approach designed was proved to be effective in Android integration mutation testing with MuDroid, limitations do exist. First and foremost, the approach could not be involved as a part of Android APK compilation since it does not require the source code. As a result, the compiled APK file needs to be unpacked and packed again for mutants generation, which increased the cost to generate mutants. Secondly, the approach targets at Smali code instead of manipulating Android bytecodes directly. Since Smali is a translation of Android bytecode in a more readable format, manipulating Smali code to affect bytecode will not be as efficient as manipulating bytecode directly. Moreover, the current way of executing test cases by simulating user behaviours was purely based on *adb*. Hence, the efficiency of this step was seriously affected by the speed of *adb* command execution.

Apart from the design limitations, the implementation of MuDroid brings some new constraints. The generation of test cases can only be achieved either pure manually or automatically in a random fashion. Due to the lack of association of MuDroid and other automated testing frameworks, the current range of acceptable type of test cases is very narrow. Furthermore, the existing way of executing test cases has no feedback in monitoring the completion of test cases. As a result, delays are required in between the execution of each test cases to make sure the test was completed before executing the next one. The threshold of this delay was affected by the speed of Android device and the complexity of the tested app.

Last but not least, due to the time limit, we only selected four subjects for the experiments in the evaluation step, not enough subjects were studied during the conduction of experiments in the evaluation step. Hence, the evaluation results may not be general to represent for any apps in the Android app store.

6.6 Future Work

MuDroid was initially designed to prove the concept of Mutation testing in Android integration level with Smali code mutation. However, it was actually designed and implemented as a framework for developers or testers to use in practice instead of being a simple prototype to assist research. Due to the time constraint, MuDroid was not developed perfectly; a few improvements still need to be made in the future.

Currently, MuDroid only triggers manually generated test cases in its self-defined format. One possible extension is to add support for other automated testing frameworks like *appium* and *Calabash* – which could also capture screenshots during the test – to extend the range of tests type MuDroid accepts. Moreover, MuDroid could extend the killing criteria by combining reading text output to the existing killing strategy. This could reduce the equivalent mutants which cannot get detected at the UI level.

More mutation operators could also be added as MuDroid only uses six mutation operators. These operators could be specially designed for Android to bring more changes in the UI.

One suggestion is to define mutation operator for XML files that store the UI information for Android apps as these files were also generated during the MuDroid execution.

7 Conclusion

The aim of this project is to adapt mutation testing techniques in Android testing at integration level. The literature review of relevant aspects of mutation testing and Android testing was conducted first to discover an approach of black-box Android integration mutation testing without accessing to the source code. The study results then led to the creation of intermediate language based approach. Consequently, Smali was selected as the target language and six novel Smali specific mutation operators were proposed. Along with that, innovative screenshots based killing criteria and selection strategies were designed as well to improve the efficiency of the approach proposed. MuDroid was then designed and developed as an implementation of the concepts proposed. Finally, experiments were designed and conducted to evaluate the efficacy, effectiveness and efficiency of the approach discovered.

This novel approach has been proved to be effective enough in gauging the quality of integration test cases with four Android apps, given that each app was tested with two test suites. Screenshot killing criteria designed was evaluated with the same test subjects. The result showed it killed about 40% more mutants than that archived by traditional crash-based killing criteria. Furthermore, this screenshots killing approach may be more effective in killing mutants for UI sensitive apps. Selection strategies proposed were evaluated with the same apps as well. During the experiment, 5000 tests were triggered for selection strategies with random factors. The result revealed Equalization Selection is more efficient in most cases with a reduction of 80% mutants and 7.3% error in average.

To summarise, the approach proposed is considered to be effective and efficient enough in executing mutation testing for Android integration testing. MuDroid framework was developed as an implementation of all the proposed concepts. This framework is so far the only intermediate language based framework for Android mutation testing in the market. It is also the only available framework specially designed for Android integration testing. We believe MuDroid could be a useful tool in determining the fault detection ability for Android integration test suites by extending the current support range of test suites. All in all, the project achieved its goals and provided a useful methodological advancement over Android mutation testing in integration level.

References

[1] (2015). Ericsson mobility report, [Online]. Available: http://www.ericsson.com/res/docs/2016/mobility-report/ericsson-mobility-report-feb-2016-interim.pdf.

[2] Statista, *Number of apps available in leading app stores as of july 2015.* [Online]. Available: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[3] R. Demillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer", *Computer*, vol. 11, no. 4, pp. 34–41, 1978, ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136.

[4] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation", in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, Szeged, Hungary: ACM, 2011, pp. 212–222, ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025144. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025144.

[5] M. Patrick and Y. Jia, "Kernel density adaptive random testing", in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, 2015, pp. 1–10. DOI: 10.1109/ICSTW.2015.7107451.

[6] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation", in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '15, Madrid, Spain: ACM, 2015, pp. 1375–1382, ISBN: 978-1-4503-3472-3. DOI: 10.1145/2739480.2754648. [Online]. Available: http://doi.acm.org/10.1145/2739480.2754648.

[7] Y. Jia, F. Wu, M. Harman, and J. Krinke, "Genetic improvement using higher order mutation", in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Companion '15, Madrid, Spain: ACM, 2015, pp. 803–804, ISBN: 978-1-4503-3488-4. DOI: 10.1145/2739482.2768417. [Online]. Available: http://doi.acm.org/10.1145/2739482.2768417.

[8] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique", in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 936–946. DOI: 10.1109/ICSE.2015.103.

[9] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence", in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 919–930, ISBN: 978-1-4503-2756-5. DOI: `10.1145/2568225.2568265`. [Online]. Available: `http://doi.acm.org/10.1145/2568225.2568265`.

[10] A. T. Acree Jr., "On mutation", AAI8107280, PhD thesis, Atlanta, GA, USA, 1980.

[11] T. A. Budd, "Mutation analysis of program test data", AAI8025191, PhD thesis, New Haven, CT, USA, 1980.

[12] A. P. Mathur and W. E. Wong, *An empirical comparison of mutation and data flow based test adequacy criteria*, 1993.

[13] W. E. Wong, "On mutation and data flow", UMI Order No. GAX94-20921, PhD thesis, West Lafayette, IN, USA, 1993.

[14] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King, "An extended overview of the mothra software testing environment", in *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, 1988, pp. 142–151. DOI: `10.1109/WST.1988.5369`.

[15] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing", *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 685–718, Jun. 1991, ISSN: 0038-0644. DOI: `10.1002/spe.4380210704`. [Online]. Available: `http://dx.doi.org/10.1002/spe.4380210704`.

[16] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators", *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996, ISSN: 1049-331X. DOI: `10.1145/227607.227610`. [Online]. Available: `http://doi.acm.org/10.1145/227607.227610`.

[17] Y. Jia and M. Harman, "Higher order mutation testing", *Information and Software Technology*, vol. 51, no. 10, pp. 1379 –1393, 2009, Source Code Analysis and Manipulation, {SCAM} 2008, ISSN: 0950-5849. DOI: `http://dx.doi.org/10.1016/j.infsof.2009.04.016`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0950584909000688`.

[18] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo, “Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation”, in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14, Vasteras, Sweden: ACM, 2014, pp. 397–408, ISBN: 978-1-4503-3013-8. DOI: `10.1145/2642937.2643008`. [Online]. Available: `http://doi.acm.org/10.1145/2642937.2643008`.

[19] W. E. Howden, “Weak mutation testing and completeness of test sets”, *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371–379, 1982, ISSN: 0098-5589. DOI: `10.1109/TSE.1982.235571`.

[20] Google, *Ui/application exerciser monkey*. [Online]. Available: `http://developer.android.com/tools/help/monkey.html`.

[21] *Androidripper*. [Online]. Available: `https://github.com/reverse-unina/AndroidRipper`.

[22] *Swifthand*. [Online]. Available: `https://github.com/wtchoi/SwiftHand`.

[23] *Calaba.sh*. [Online]. Available: `http://calaba.sh/`.

[24] *Selendroid*. [Online]. Available: `http://selendroid.io/`.

[25] *Appium*. [Online]. Available: `http://appium.io/`.

[26] D. Ehringer, “The dalvik virtual machine architecture”, 2010.

[27] *Mujava*. [Online]. Available: `https://cs.gmu.edu/~offutt/mujava/`.

[28] *Pit mutation testing*. [Online]. Available: `http://pitest.org/`.

[29] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, “Towards mutation analysis of android apps”, in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, 2015, pp. 1–10. DOI: `10.1109/ICSTW.2015.7107450`.

[30] *Smali*. [Online]. Available: `https://github.com/JesusFreke/smali`.

[31] Google, *Android debug bridge*. [Online]. Available: `http://developer.android.com/tools/help/adb.html`.

[32] *Apktool*. [Online]. Available: `http://ibotpeaches.github.io/Apktool/`.

[33] ——, *Adb shell commands*. [Online]. Available: `http://developer.android.com/tools/help/shell.html`.

[34] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps", in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hong Kong, China: ACM, 2014, pp. 599–609, ISBN: 978-1-4503-3056-5. DOI: `10.1145/2635868.2635896`. [Online]. Available: `http://doi.acm.org/10.1145/2635868.2635896`.

[35] *F-droid.* [Online]. Available: `https://f-droid.org/`.

[36] *Agram repository.* [Online]. Available: `https://github.com/achromaticmetaphor/agram`.

[37] *Agram.* [Online]. Available: `https://f-droid.org/repository/browse/?fdcategory=Games&fdid=us.achromaticmetaphor.agram`.

[38] *Clean calculator repository.* [Online]. Available: `https://github.com/jchmrt/clean-calculator`.

[39] *Clean calculator.* [Online]. Available: `https://f-droid.org/repository/browse/?fdfilter=calculator&fdid=home.jmstudios.calc`.

[40] *Did i ?* [Online]. Available: `https://f-droid.org/repository/browse/?fdfilter=didi&fdid=si.modrajagoda.didi`.

[41] *Droid draw repository.* [Online]. Available: `https://bitbucket.org/XatikGroup/droiddraw/`.

[42] *Droid draw.* [Online]. Available: `https://f-droid.org/repository/browse/?fdfilter=droiddraw&fdid=com.xatik.app.droiddraw.client`.

[43] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing", *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011, ISSN: 0098-5589. DOI: `10.1109/TSE.2010.62`.

[44] M. Usaola and P. Mateo, "Mutation testing cost reduction techniques: A survey", *IEEE software*, vol. 27, no. 3, p. 80, 2010.

[45] P. E. Black, V. Okun, and Y. Yesha, "Mutation operators for specifications", in *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, 2000, pp. 81–88. DOI: `10.1109/ASE.2000.873653`.

[46] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, “An experimental mutation system for java”, *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–4, Sep. 2004, ISSN: 0163-5948. DOI: 10.1145/1022494.1022537. [Online]. Available: http://doi.acm.org/10.1145/1022494.1022537.

[47] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, “Inter-class mutation operators for java”, in *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, 2002, pp. 352–363. DOI: 10.1109/ISSRE.2002.1173287.

Appendix

A Source Code

MuDroid is available in GitHub at: https://github.com/Yuan-W/muDroid The web sites have all the MuDroid source code files and instructions for how to and use MuDroid.

B Evaluation Result

Due to the screenshot-based killing criteria, the evaluation result for MuDroid contains 65,41 file and takes 6.23GB in total. It is not possible to attach such a large amount of files or event put them online. Hence, only the summarised report for each evaluation subject are available at: https://goo.gl/A9Qc2o

C Mutation Tester

Mutation Tester is an Android app specially designed to test MuDroid in the integration level. Its source code and the test coverage report are available at: https://goo.gl/GS8YWt

D Project Plan

MuDroid: Mutation Testing for Android Apps

Project Plan

Prepared By:
Yuan Wei
(SN:13022194)

Supervised By:
Yue Jia

Introduction

According to the latest annual mobility report from Ericsson, there're about 2.6 billion smart phone users in the world. Each mobile app user could spends about 3 hours per day on the phone. A recent report also shows the total revenue for Google Play and iTunes App store is about $23 billions on 2014. Mobile apps were deeply involved in people's daily life. It is important for app developers to assure their apps work as expected, otherwise it will troubles there millions of users.

Mobile phone testing is vital. Not all user will using the app in the way developers expected. With the help of modern techniques, developers could develop an app in few weeks or even few days. It is difficult for developers to find out all the bugs in the app in such a short time. Therefore, without proper testing, users will come across many unexpected bugs or errors. But that also brings a new questions: How to find the completion criteria that stops the testing? Mutation testing is one of the answers to the question.

Mutation testing is a technique that seed mutants into the code, then gauge the quality of tests based the percentage of mutants being killed. Compering with traditional 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. It is capable of simulating the effect of other white box testing techniques and providing improved fault detection through automated fault injection.Thus, it's a better standard in finding the completion criteria stops the testing.

Aim

Although mutation testing has been proved to be an effective way to test programs, to date, less work has been focused on testing mobile apps. Also, existing mutation testing projects only focus generic unit testing. However, when it come to mobile apps testing, people are more interested in integration testing. Instead of create unit test for the backend code, integration testing for mobile apps targets the integration between user interface and the backend code. That could be a more challenging scenario as it requires a different form of testing and result collection. This project proposes to develop MuDroid, a mutation testing tool for Android testing at integration level.

Objectives

1. Define mutation operators for Android specific features.
2. Development mutation environment for Android platform.
3. Implement the mutation operators defined above for Android apps.
4. Evaluate the implementation with monkey testing on real world Android apps.

Deliverables

1. Mutation operators defined for Android specific feature.
2. MuDroid, a mutation system could seed mutations using the operator defined above into Android apps and generate report based on the result collected.

Work Plan

Project start to end October (4 weeks)

Literature search and review on mutation testing.
Set up the development and testing environment.

November (4 weeks)

Modelling and prototyping of the project.
Define mutation operators for Android testing.
Develop the prototype iteratively.

December (4 weeks)

Develop the prototype iteratively.
Improve the software for report generation.
Add more mutation operators.

January (4 weeks)

Develop the prototype iteratively.
Test the project with android apps from Android market.

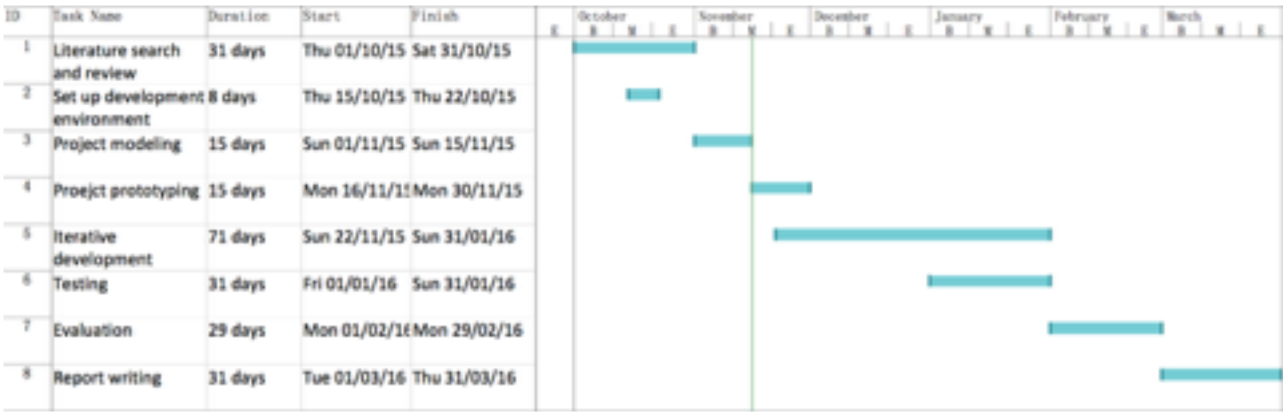
February (4 weeks)

Evaluation.

March (4 weeks)

Final report writing.

Gantt Chart



ID	Task Name	Duration	Start	Finish
1	Literature search and review	31 days	Thu 01/10/15	Sat 31/10/15
2	Set up development environment	8 days	Thu 15/10/15	Thu 22/10/15
3	Project modeling	15 days	Sun 01/11/15	Sun 15/11/15
4	Proejct prototyping	15 days	Mon 16/11/1!	Mon 30/11/15
5	Iterative development	71 days	Sun 22/11/15	Sun 31/01/16
6	Testing	31 days	Fri 01/01/16	Sun 31/01/16
7	Evaluation	29 days	Mon 01/02/16	Mon 29/02/16
8	Report writing	31 days	Tue 01/03/16	Thu 31/03/16

E Interim Report

MuDroid: Mutation Testing for Android Apps

Interim Report

Prepared By:
Yuan Wei
(SN:13022194)

Supervised By:
Yue Jia

Introduction

According to the latest annual mobility report from Ericsson, there are about 2.6 billion smartphone users in the world. Each mobile app user could spend about 3 hours per day on the phone. A recent report also shows the total revenue for Google Play and iTunes App store is about $23 billions on 2014. Mobile apps were deeply involved in people's daily life. It is important for app developers to assure their apps work as expected. Otherwise, it will trouble these millions of users.

Mobile phone testing is vital. Not all user will use the app in the way developers expected. With the help of modern techniques, developers could develop an app in few weeks or even few days. It is difficult for developers to find out all the bugs in the app in such a short time. Therefore, without proper testing, users will come across many unexpected bugs or errors. However, that also brings a new questions: How to find the completion criteria that stops the testing? Mutation testing is one of the answers to the question.

Mutation testing is a technique that seed mutants into the code, then gauge the quality of tests based the percentage of killed mutants. Comparing with traditional techniques like line coverage which only measures the percentage of executed code, mutation testing could identify the ability of fault detection for tests. It is capable of simulating the effect of other white box testing techniques and providing improved fault detection through automated fault injection.Thus, it is a better standard in finding the completion criteria stops the testing.

Aim

Although mutation testing has been proved to be an effective way to test programs, to date, less work has been focused on testing mobile apps. Also, existing mutation testing projects only focus generic unit testing. However, when it come to mobile apps testing, people are more interested in integration testing. Instead of creating unit tests for the backend code, integration testing for mobile apps targets the integration between the user interface and the backend code. That could be a more challenging scenario as it requires a different form of testing and result collection. This project proposes to develop MuDroid, a mutation testing tool for Android testing at integration level.

Project Progress

After few months' research and development, a simple prototype of MuDroid was created. It could mutate an existing Android apk file with pre-defined mutation operator and generate a simple report. In details, the prototype could generate random screen click and swipe events with a given number. It then analysis the apk file and generate mutants, each with an individual apk file. After this, it triggers an interaction testing with events generated in the previous step for each mutant. After each click or swipe event, screenshots were captured. Screenshots from mutants will be compared with original program's screenshot after each interaction testing to identify whether the mutant is killed or not. In the end, a simple HTML format report with all mutants' status is generated. The prototype is a proof of concept and could archive the minimum request of the original goal. Future work will be interactively developed based on this prototype.

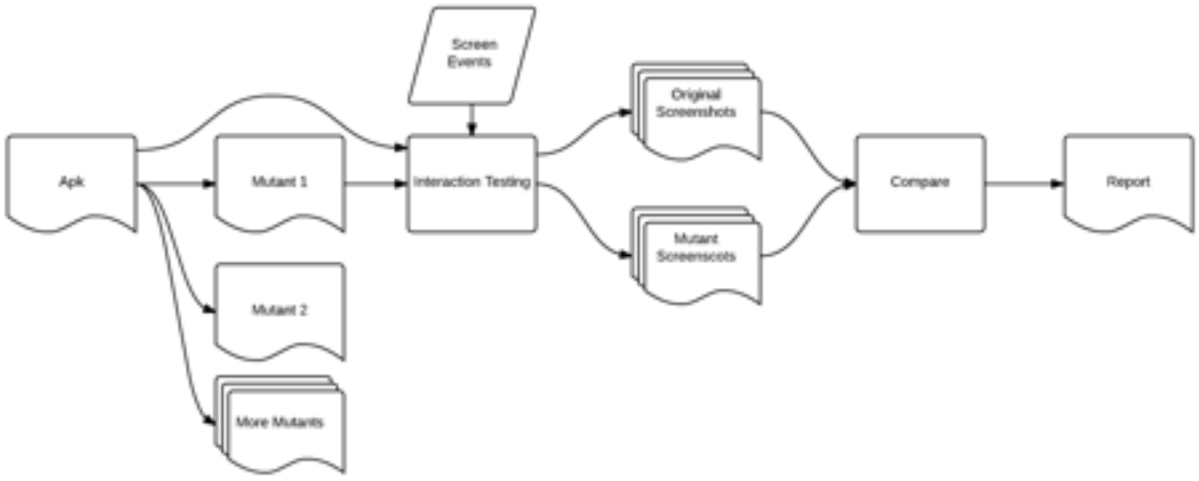


Figure 1: Project Workflow

Remaining Work

1. Implement more mutation operators. For now, only AOR (arithmetic operator replacement) is implemented. More operator needs to implement. Offutt et al.'s research, ABS, UOI, LCR, AOR, and ROR are the five key mutation operators and could achieve 99.5% mutation score.
2. Improve the quality of generated report to make it more readable and provides more information.
3. Test and fix bugs for the prototype with more apk files from Android market.
4. Project Evaluation.