REGRESSION TEST SELECTION FOR ANDROID APPLICATIONS

by

Quan Chau Dong Do, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Software Engineering
December 2015

Committee Members:

Guowei Yang, Chair

Rodion Podorozhny

Anne Ngu

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

...this research is dedicated to my wife, my resource of encouragement

ACKNOWLEDGEMENTS

I am thankful to Dr. Guowie Yang for his practical advice and his criticism. He sets high standard for students in classrooms as well as in research practice. I am grateful to him for holding such a research standard from proposing a project to enforce strict validation for research project. I would not be able to finish this project without his advice.

I would like to thank my thesis committee. Dr. Rodion Podorozhny and Dr. Anne Ngu who have been great professors help me prepare to get to this place in my academic life.

Most importantly, none of this would have been possible without the love, support and encouragement of my wife, and my parents who live thousand miles away.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF GRAPHS

Graph **Page**

LIST OF ABBREVIATIONS

Abbreviation	Description
Davik Virtual Machine	DVM
Java Virtual Machine	JVM
Control Flow Graph	CFG
Impact Analyzer	IA
Test Case Selector	TCS
Code Coverage Generator	CCG
Application Under Test	AUT
Anonymous Inner Class	AIC
Retest All Strategy	RAS
Test Selection Framework for Android apps	TSFA
Regression test selection	RTS

ABSTRACT

As the mobile platform pervades human life, much research in recent years has focused on improving the reliability of mobile applications on this platform, for example by applying automatic testing. To date, however, researchers have primarily considered testing of the single version of mobile applications. It has been shown that testing of mobile applications can be expensive; thus simply re-executing all tests on the modified application version remains challenging. Regression testing---a process of validating modified software to ensure that the changes are correct and do not adversely affect other features of the software---has been extensively studied for desktop application, and many efficient and effective approaches have been proposed; however, these approaches cannot be directly applied to mobile applications. Since regression testing on mobile applications is an expensive process, an effective and well-studied regression test selection can potentially reduce this expense. In this study, we propose test selection for mobile applications, especially on the Android Application Platform. Our approach leverages the combination of static impact analysis with code coverage that is dynamically generated at run-time, and identify a subset of tests to check the behaviors of the modified version that can potentially be different from the original version. We implement our approach for Google Android applications, and demonstrate its effectiveness using an empirical study.

1. INTRODUCTION

Mobile devices have become ubiquitous in modern society. The mobile platform is separating itself from a variety of areas of desktop applications such as entertainment, e-commerce and social media. Thus, developers are required to produce high quality mobile apps in terms of portability, reliability and security. In recent years, a large number of research projects have focused on improving the reliability of mobile applications on mobile platform, for example by applying automatic testing. However, the majority of the researches is only focusing on testing of the single version of mobile applications. It has been shown that testing of mobile applications is not a trivial task and can be expensive. As developers periodically maintain a software system, they perform regression testing to find errors that are caused by program changes and to provide confidence that the modifications are correct. To support this process, according to Todd et al (2001), developers often create an initial test suite, and then they reuse it to perform regression testing. Regression testing---a process of validating modified software to ensure that the changes are correct, and they do not adversely affect other features of the software---has been extensively studied for desktop applications. Many efficient and effective approaches have been proposed; however, these approaches cannot be directly applied to mobile applications. One of the factors that causes the incompatibility is the difference between the mobile platform system architectures and the desktop platform. For example, although Android Platform is written in Java as other Java desktop applications, it runs under the Dalvik Virtual Machine (DVM). The DVM and the Java byte-code run-time environment are substantially different, according to Williams (2011).

The traditional approach is known as the Retest All Strategy (RAS). RAS is simply to re-execute every single test case in the original test suite. However, this approach can be expensively unacceptable, since it requires a tremendous amount of time, especially on mobile apps. Since performing regression testing on mobile applications by applying RAS is an expensive process, an effective and well-studied regression test selection can potentially reduce this expense. In this study, we propose a regression test selection technique for mobile applications, especially on the Android Application Platform. Our approach, a Test Selection Framework for Android apps (TSFA), identifies which subset of a test suite must be re-executed to test a new version of an Android application. TSFA leverages the combination of static impact analysis and dynamic analysis. It first, detect changes between the original version and the modified version of the program. Then, TSFA will dynamically generate code coverage for each test case. Lastly, our approach will select a subset of the test suite for re-execution in order to check the behaviors of the modified version that can potentially be different from the original version. We implement our approach for Google Android applications, and demonstrate its efficiency and effectiveness using an empirical study.

2. MOTIVATING EXAMPLE

In this chapter, we use a simple Android application to illustrate the goal of this study. Note that we developed this application and wrote a test suite to test it. We also purposely made changes to specific locations of the modified program. This example is designed for illustration only, not for the determination of the overall performance of our

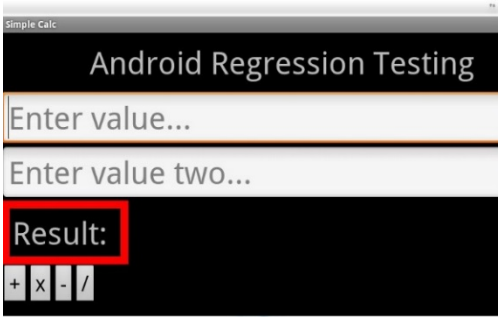


Figure 1: Simple Calculator Android Application

study. Figure 1 is a screenshot of the *Simple Calculator Application,* a basic calculator that takes as input two numbers and performs different arithmetic including addition, subtraction, multiplication and division by clicking buttons *"+", "-", "×",* and *"/"* respectively. Figure 2 shows the implementations of four buttons. Let *T* be a test suite consisting of 18 test cases as described in Figure 3. After executing the test suite, based on Figure 2, two bugs are found at line number 4 and 22. In terms of maintenance, after modifying an implementation, software is required to be re-tested in order to assure that changes do not adversely affect other software components. For the traditional regression testing, the entire test suite is rerun against the modified code to provide confidence that the changes are correct. For the Android Platform, this approach remains challenging due to the cost of executing a large number of test cases. Therefore, reducing the number of test cases is significantly important. Instead of re-executing the entire test suite, TSFA selects the test cases that are affected by the program changes. For example, *T* is used to

test the application. Each test case takes as input two numbers and presses the arithmetic buttons accordingly. After executing the test suite, a code coverage report for each test case is generated. For example, the code coverage report for T_1, T_2, and T_3 is 1->6, which indicates the source code line number 1 to 6 as described in Figure 2.

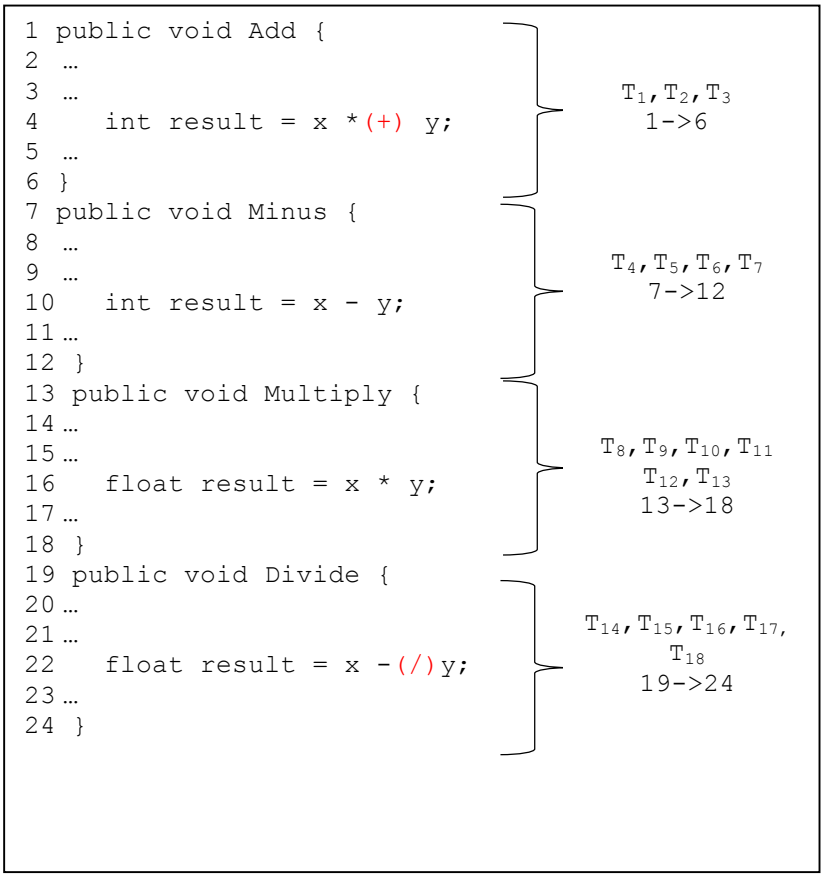


Figure 2: Implementations of Four Arithmetic Methods

Let *P* be the original program. *P* is modified at line number 4 (x*y becomes x+y) and at line number 22 (x-y becomes x/y). Let *P'* be the modified program. After the modifications, it is required that only a subset, *T'* in *T* is selected for re-execution. From the code coverage reports, let $T\sim$ be a set of test cases that are not affected by the changes, it is easy to show that $T\sim=\{T_4,T_5,T_6,T_7,T_8,T_9,T_{10},T_{11},T_{12},T_{13}\}$ is not eligible for re-execution. Only a subset of *T*, that is, $T'=\{T_1,T_2,T_3,T_{14},T_{15},T_{16},T_{17},T_{18}\}$ is selected

for re-execution since the test cases in *T'* executed the changes. As the result, only eight test cases are selected. Suppose each test case takes five seconds to be executed, then it would take 90 seconds when applying RAS. However, it only takes 40 seconds when applying TSFA since only 45% of the test cases are selected.

testT1(){
enterText("3","4");
clickOnButton("+");
result = getView(id.result);
assertEquals(T1_RESULT,result);
}
	testT2(){
	enterText("-2","7");
	clickOnButton("+");
	result = getView(id.result);
	assertEquals(T2_RESULT,result);
	}

testT3(){
enterText("-1","-4");
clickOnButton("+");
result = getView(id.result);
assertEquals(T3_RESULT,result);
}
	testT4(){
	enterText("0","2");
	clickOnButton("-");
	result = getView(id.result);
	assertEquals(T4_RESULT,result);
	}

testT5(){
enterText("-4","-5");
clickOnButton("-");
result = getView(id.result);
assertEquals(T5_RESULT,result);
}
	testT6(){
	enterText("3","7");
	clickOnButton("-");
	result = getView(id.result);
	assertEquals(T6_RESULT,result);
	}

testT7(){
enterText("9","2");
clickOnButton("-");
result = getView(id.result);
assertEquals(T7_RESULT,result);
}
	testT8(){
	enterText("0","1");
	clickOnButton("x");
	result = getView(id.result);
	assertEquals(T8_RESULT,result);
	}

testT9(){
enterText("-2","-3");
clickOnButton("x");
result = getView(id.result);
assertEquals(T9_RESULT,result);
}
	testT10(){
	enterText("3","-1");
	clickOnButton("x");
	result = getView(id.result);
	assertEquals(T10_RESULT,result)
	}

testT11(){
enterText("9999999","99999");
clickOnButton("x");
result = getView(id.result);
assertEquals(T11_RESULT,result)
}
	testT12(){
	enterText("-4","5");
	clickOnButton("x");
	result = getView(id.result);
	assertEquals(T12_RESULT,result)
	}

public void testT13(){
enterText("0","0");
clickOnButton("x");
result = getView(id.result);
assertEquals(T13_RESULT,result)
}
	public void testT14(){
	enterText("0","0");
	clickOnButton("/");
	result = getView(id.result);
	assertEquals(T14_RESULT,result)
	}

public void testT15(){
enterText("2","2");
clickOnButton("/");
result = getView(id.result);
assertEquals(T15_RESULT,result)
}
	public void testT16(){
	enterText("2","3");
	clickOnButton("/");
	result = getView(id.result);
	assertEquals(T16_RESULT,result)
	}

public void testT17(){
enterText("1","99999999");
clickOnButton("/");
result = getView(id.result);
assertEquals(T17_RESULT,result)
}
	public void testT18(){
	enterText("-2","4");
	clickOnButton("/");
	result = getView(id.result);
	assertEquals(T18_RESULT,result)
	}

Figure 3: Example of 18 Test Cases

3. BACKGROUND

This chapter provides background knowledge that is relative to our study. First, we summarize the architecture of developing Android applications. Next, we provide a brief definition of a Control Flow Graph and we show how it is constructed for Android applications. Lastly, we recall the concept of regression testing to which the TSFA is related.

3.1 Android Development Platform

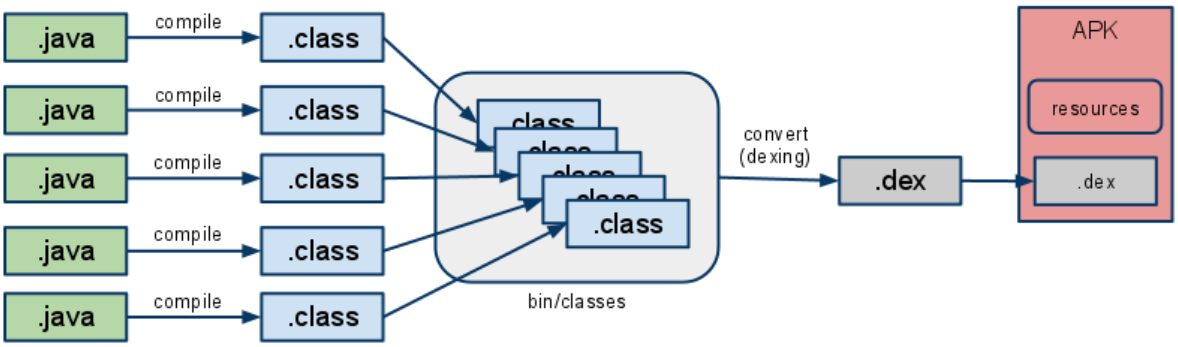


Figure 4: Android Development Architecture

Each Android application is executed under the Davik Virtual Machine (DVM) and Android applications are written in the Java programming language. As described in Figure 4, first, the Java source code files are compiled to bytecode class files using standard Java Virtual Machine (JVM). After the compilation, the Davik *dx* tool converts the class files to Davik bytecode file, and composes them into one single dex file. The dex file contains all of the application classes. Finally, in order to run on an Android device, the dex file and resources are composed into one executable *apk* file and the Android device proceeds installation using the *apk* file. Our approach assumes that application's source code is always available.

We leverage the analysis of Java byte code to implement our framework, which is divided into two main components. Firstly, we generate code coverage for each test case that is executed by the application. Before the Davik dx tool coverts the class files into dex file, we perform code instrumentation for the source code. Then, the final apk file will contain the instrumentation as it is being used to install the application. Secondly, we analyze the impact of each version of the application using the available source code.

3.2 Construction of Control Flow Graph of Java bytecode

Definition 3.2.1. Directed Graphs

A directed graph is a graph $G = (N, E)$ with a set of nodes N that are connected by a set of edges E, with functions start and end. We denote (n, n') is an edge starts from node *n* and ends at node n' with $n, n' \in N$. For a node *n,* the sets

$pred(n) = \{n' \mid \exists e \in E : start(e) = n', end(e) = n\}$ and the sets

$succ(n) = \{n' \mid \exists e \in E : start(e) = n, end(e) = n'\}$ contain all of predecessors and successors of *n,* respectively. *Entry nodes* and *exit nodes* are nodes that have an empty *pred* or *succ* set, respectively.

Definition 3.2.2. A Control Flow Graph

A Control Flow Graph *(CFG)* is a representation of a directed graph. In a procedure of a programming language, each statement in a procedure will be a node in the CFG. The control flow is represented by the edges. An entry node and exit node are uniquely added to the starting and ending point of the procedure.

A node *n* in a Control Flow Graph for Java Bytecode has multiple bytecode statements since it represents a basic block of a procedure *p.* A node $\mathrm{n}_{s,e} \in N$ indicates a

basic block that has starting statement s and ending statement e. A node $n_{s,s}$ indicates a block that has one statement s. An edge $(n_{e,s}, n_{e',s'})$ is created when *s* is a branching instruction and *e'* is its target. Entry node and exit are uniquely added to CFG as $n_{-1,-1}$ and $n_{-2,-2}$, respectively. Figure 5 shows an example of a CFG for a simple if statement.

3.3 Regression Testing

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is performed between P and P' to ensure that changes in P' do not affect P by executing T against P'. However, executing T can be unnecessarily expensive, especially when only a small part of the system is affected. Rothermel G. and Harrold M. J. (1996) have studied several regression test selections (RTS).

ByteCode	CFG

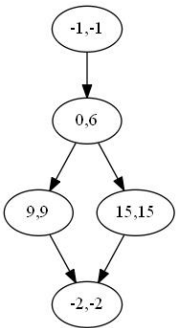
main(){
0: iconst_0
1: istore_1
2: iconst_1
3: istore_2
4: iload_1
5: iload_2
6: if_icmple 15
9: iinc 1, 1
12: goto 18
15: iinc 2, 1
18: return
}
	

Figure 5: Examples of CFG

Retest All is the technique that re-executes all test cases. It can be applied to situations where no cost of execution time is being concerned.

Random/Ad-Hoc technique randomly selects a percentage of test cases. If fault detection is not concerned, this technique can reduce a great amount of execution time.

Minimization technique has been studied by Fischer K., Raji F., and Chruscicki A. (1981) which selects a minimal subset of a test suite that covers all program changes.

Safe technique, according to Rothermel G. and Harrold M. J. (1997), under certain conditions, selects a test case when it covers at least one change in the modified program.

Selecting $T' \subseteq T$ as a set of test cases to execute on P' will efficiently reduce the execution time. Hiralal A., Joseph R. H., Edward W. K., Saul A. L. (1993) observed if a changed statement is not executed by a test case, the program output for that test cannot be affected. Hence, code coverage for each test case is required in order to determine which statement is not executed by a test case. We leverage the Safe technique as the foundation of the TSFA. According to the study of Harrold M. J., and Rothermel G. (1997), in terms of fault the detection effectiveness, Safe technique performs slightly better than random technique. Even though, in some cases such as low code coverage, Safe technique is not efficient in reducing execution time, it is very effective in fault detection. Figure 6 shows an example of the Safe technique. Given a CFG with 10 nodes. The dash, solid and dot lines represent the code coverage Test 1, Test 2 and Test 3, respectively. Suppose a change is found at Node 4. Then only Test 3 is selected for re-execution. Since Test 1 and Test 2 do not execute the change, they are eliminated. According to Todd L. G., Mary J. H., Jung-Min K., Adam P., Gregg R. (2001), this approach is not always perfect. In other words, this test case selection technique substantially reduces the cost of execution time while it may eliminate test cases that could reveal faults in the program, and consequently, reducing the effectiveness of fault detection. Moreover, Hiralal A., Joseph R. H., Edward W. K., Saul A. L. (1993) also

observed that not all statements in the program are executed under all test cases. A changed statement might not be executed by any test case, which might result in the elimination of all test cases. It is a trade-off between reducing execution time and decreasing the effectiveness of fault detection.

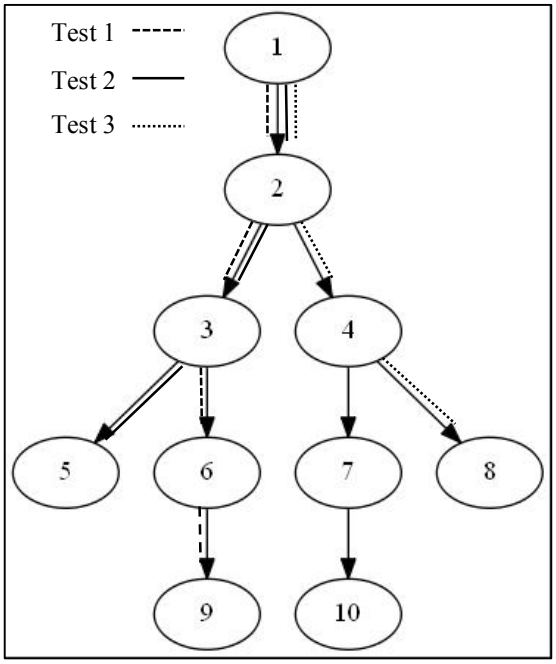


Figure 6. Safe Technique

4. APPROACH

4.1 Overview

TSFA consists of three components: *Impact Analyzer* (IA)*, Code Coverage Generator* (CCG) *and Test Case Selector* (TCS) as described in Figure 7.

Firstly, CCG takes as input the original program *P* and generates code coverage report. In this step, CCG performs code instrumentation on *P* to obtain P^T. Then, we run a test suite *T* against P^T to the collect code coverage for each test case T_i in *T*. As a test case T_i is running, our framework records code coverage from P^T.

Secondly, IA takes as input the original program *P* and its the modified version *P'*, and generates an impact report that contains locations of changes between *P* and *P'*.

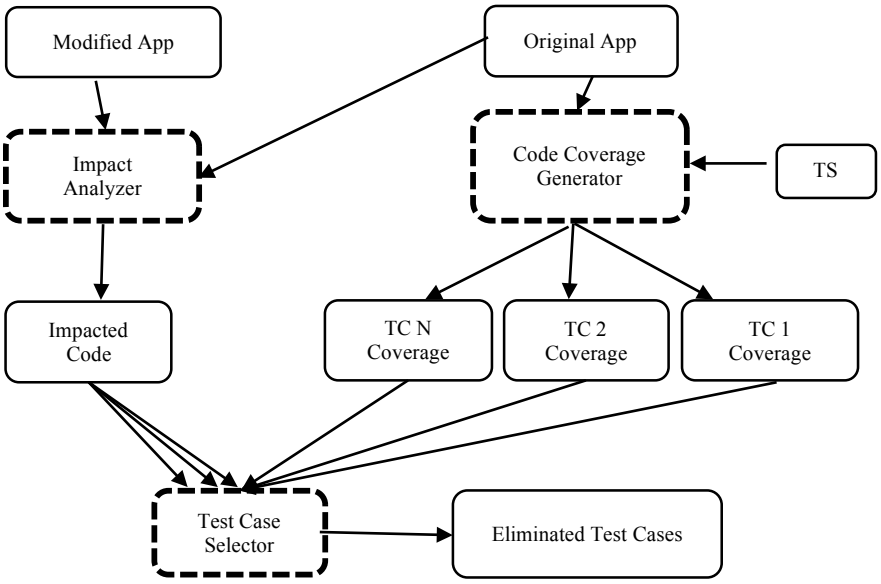


Figure 7: Approach Architecture

Lastly, TCS takes as input the impact report and the code coverage report to determine which test cases are affected by the changes and select them. As a result, only a subset T' of T is selected for re-execution.

4.2 Impact Analyzer

The IA takes as input an original class file C and a modified class file C'. Firstly, it builds a CFG and CFG' for C, and C', respectively. Each CFG represents a procedure M. Then, IA uses the depth-first search concept to compare C and C' node by node, and instruction by instruction. The impact report is stored as a text file whose each line indicates a location of a basic block where the change occurs. A location of a basic block is formatted as the following:

$$name\ of\ class, name\ of\ method, first\ bytecode\ statement\ of\ the\ changed\ block$$

For example,

$$org.andstatus.app.HelpActivity, get, 39: aload\_1[43](1)$$

In this example, in the class $HelpActivity$, at method $get()$, the block that has a starting bytecode statement as $39: aload\_1[43](1)$ has changed in P'.

4.3 Code Coverage Generator

CCG leverages the combination of static code analysis and dynamic analysis. It first performs code instrumentation on the original program *P*. Then, it executes a test suite to collect code coverage for each test case.

When building a CFG for each method of *P*, before forming edges between nodes, the code instrumentation framework inserts new bytecode statements into the instruction list. Figure 8 shows the code instrumentation algorithm: *instrumentCCG*. Let *I* be a set of all bytecode instructions in procedure *p*. A node $n_{l,e} \in N$ indicates a basic block that has a starting leading bytecode instruction *l* and an ending bytecode instruction *e*. Let *L* be a set of all leading bytecode instructions in *p*. When inserting new bytecode

block $n'_{l,e} \in N$, we choose the beginning of each basic block as the inserted position. Consequently, *L* now no longer contains the original leading bytecode instructions. In order to preserve the original L, we introduce Map_L, a set of maps such that each map ($l \rightarrow l'$) represents a map from an old leading instruction *l* to a new leading instruction *l'*. The method *getNewLeader()* returns *l'* by getting the leading instrument of $n'_{l,e}$. A program can contain branching instructions such as if-statements, case statements, loops, etc. If an instruction $b \in l$ is a branching instruction, there exists a target instruction $t \in I$ such that a target map ($b \rightarrow t$) represents its branching relationship. As a matter of fact, b is the ending bytecode instruction of one block, and t becomes a leading bytecode instruction of the target block. Let Map_T be a set of target maps. The method *addNewBytecode()* takes as input the current leading instruction bytecode and inserts new block of bytecode on top of it. Given a set of maps $M = \{(a \rightarrow b)\}$, for map $(a \rightarrow b)$, $M.key$ returns *a* and $M.value$ returns *b.*

In the *instrumentCCG* algorithm, from line 1 to line 5, new block of bytecode instructions are inserted at the beginning of every $l \in L$. Since every $l \in L$ is now no longer a leading bytecode instruction, Map_L stores maps from old leading instructions to new leading instructions at line 5. A target map ($b \rightarrow t$) is also changed due to this modification since *t* becomes a leading bytecode statement of another block. From line 6 to line 9, Map_T is used to update target maps accordingly to the new leading instructions. At line 8 and 9, for every *t* that is equal to *l,* a new value *t'* replaces *t.* Up to this point, the actual target instructions in *I* have not yet been updated. From line 10 to line 13, for every

branching instruction $b \in I$, b is updated accordingly using Map_T. Figure 9 shows an example of instrumented bytecode.

CCG uses the Log system of the Android platform to generate a log file that contains code coverage information. Specifically, $Log.d(String\ tag, String\ message)$ statement prints the message and its tag to an Android log file. As a test suite is executed against the instrumented program, by automatically instrumenting equivalent bytecode into the entire program, our framework is able to log code coverage of each test case.

```
instrumentCCG()
1   for i ∈ I
2     for l ∈ L
3       if i = l
4         addNewBytecode(i)
5           Map_L ← (ci → getNewLeader())
6   for t_M ∈ Map_T
7     for l_M ∈ Map_L
8       if t_M.value = l_M.key
9         t_M.value ← l_M.value
10  for i ∈ I
11    for t_M ∈ Map_T
12      if i = t_M.key
13        setNewTarget(i, t_M.value)
end instrumentCCG()
```

Figure 8: *intrumentCCG* Algorithm

4.4 Test Case Selector

TCS is a simple Java program that takes as input two text files, namely the impact report and code coverage report. It determines which test cases are affected by the changes and selects them for re-execution. Log file analysis process is to (1) parse the log

file, (2) store executed block locations by creating maps between test cases and their coverage, (3) store impacted block location, and (4) select test cases.

Figure 10 shows an example of the Android log file. First TCS takes as input the log file, which is created as text file format. Parsing process starts by searching line by line of the text file.

ByteCode	Instrumented Bytecode
main(){	main() **0: ldc** **2: ldc** **4: invokestatic** **7: pop**
0: iconst_0	8: iconst_0
1: istore_1	9: istore_1
2: iconst_1	10: iconst_1
3: istore_2	11: istore_2
4: iload_1	12: iload_1
5: iload_2	13: iload_2
6: if_icmple 15	14: if_icmple 31 **17: ldc** **19: ldc** **21: invokestatic** **24: pop**
9: iinc 1, 1	25: iinc 1, 1
12: goto 18	28: goto 42 **31: ldc** **33: ldc** **35: invokestatic** **38: pop**
15: iinc 2, 1	39: iinc 2, 1
18: return	42: return
}	}

Figure 9: Example of Instrumented Bytecode

Let Map_C be a set of maps such that each map ($T_i \rightarrow [a_1, a_2, ..., a_n]$) where T_i is the name of the test case, and $a_1, a_2, ..., a_n$ are the locations of executed blocks. They are formatted are described in Section 4.2. Let *C* be a set of changes { $c_1, c_2, ..., c_n$ } where $c_1, c_2, ..., c_n$ are affected blocks.

When a line starts with the tag *I/TestRunner: started*, it indicates that the beginning of a test case has already started. T_i is assigned a value which is the name of the test case. For example, in Figure 10, at line 1, T_i = *testWifiConnected*. Every next line that starts with the tag *D/INSTRUMENT* indicates the executed block location. $a_1, a_2, ..., a_n$ are assigned values corresponding to these lines. For example, at line 2 and 3, a_1=*ConnectivityActionReceiver,onReceive,0: aload_1[43](1)* and a_2=*ConnectivityActionReceiver,onReceive,20: aload_2[44](1)*, respectively. When a line with the tag *I/TestRunner: finished,* it indicates the completion of the test case. Then, a map $T_i \rightarrow [a_1, a_2]$ is added to Map_C. This process is repeated to end of the log file.

```
1. I/TestRunner: started: testWifiConnected
2. D/INSTRUMENT: ConnectivityActionReceiver,onReceive,0: aload_1[43](1)
3. D/INSTRUMENT: ConnectivityActionReceiver,onReceive,20: aload_2[44](1)
4. D/INSTRUMENT: ConnectivityActionReceiver,onReceive,27: iload[21](2) 4
5. I/TestRunner: finished: testWifiConnected
6. I/TestRunner: started: testAddIgnoredWifi
7. D/INSTRUMENT: DatabaseAdapterImpl,openIfNeeded,17: aload_0[42](1)
8. D/INSTRUMENT: DatabaseOpenHelper,onCreate,0: aload_1[43](1)
9. D/INSTRUMENT: DatabaseAdapterImpl,addIgnoredWifi,62: iconst_1[4](1)
10.  I/TestRunner: finished: testAddIgnoredWifi
```

Figure 10: An Example of a Log File

TCS takes as input Map_C and C to select test cases. For each $c_i \in C$, if $c_i \in [a_1, a_2, ..., a_n]$, then the corresponding T_i is selected. The actual implementation of the *instrumentCCG* is presented in the Appendix.

5. EMPIRICAL STUDY

We evaluate the efficiency and correctness of our Code Coverage Generator, by considering two research questions:

RQ1. How efficient is our Code Coverage Generator compared to an existing Code Coverage Generator, Emma?

RQ2. Does the Code Coverage Generator achieve the same level of correctness compared to Emma?

Emma, an effective code coverage generator, is being compared with our CCG because it is the only code coverage tool that is built-in for the Android Development Environment. Plus, Emma provides high precision for code coverage.

We then evaluate the overall cost of TSFA relative to RAS by considering two research questions:

RQ3. How does the number of test cases selected by TSFA compare to RAS, which re-runs all test cases?

RQ4. How efficient is the time reduction when applying TSFA compared to RAS?

5.1 Artifacts

We selected two Android applications to evaluate TSFA: *AndStatus* and *Inetify*. For each artifact, Table 1 provides information on its associated "Classes" – number of class files, "Methods" – number of methods, "LOC" – number of lines of code, and "Version" – number of modified versions we used for the evaluation.

The first artifact, the *AndStatus* application, is an Android open source project. This application allows users to login multiple social app accounts such as Twitter and Pump.io. *AndStatus* can combine multiple accounts into one Timeline and allow users to

read and post even if they are offline.

The second artifact, the *Inetify* application, is an Android open source project that provides two features related to Wifi networks. First, the app gives a notification if a Wifi network does not provide Internet access. Secondly, it automatically activates Wifi when being near a Wifi network and deactivates it otherwise.

Table 1. Artifacts Used for the Evaluation

	Classes	Methods	LOC	Version
AndStatus	250	2700	15000	15
Inetify	63	356	1500	15

We evaluate TSFA on multiple versions of each artifact in order to increase the degree of randomization when modifying the application under test (AUT) to reduce bias. When creating modified versions of the AUTs, we examine the extensive dimension of changes, including different numbers of changes, locations, and change type. We consider three possible numbers of changes to apply: one, three, and five. By simulating changes, we are able to create multiple versions for each artifact. Although the changes are not actual modifications from, for example, developers or testers, we have control over locating them in such a systematic way that large scale applications can be thoroughly evaluated. The changes include modification, deletion, and addition of source code statements. In the modified AUTs, changes are located at, (1) control statements, including if statements, case statements, and while, for, and do loops, (2) non-control statements, (3) general locations such as top, middle and bottom of the program source code, (4) Anonymous Inner Class (AIC). Different from regular Java procedures, which

are defined as a separated method, AIC is defined as an inner procedure of another procedure. Figure 11 shows an example of AIC.

The method *setOnClickListener()* is an AIC procedure defined as an inner procedure of the *onCreate()* method. We consider the case of AIC procedures because, as a result, Android Development Environment generates separate class files for them. Plus, IA takes as input class files to generate the impact report. Hence, considering changes in AIC procedures will increase the ability of finding changes in the modified AUTs. Control statements are modified by changing the comparison operator, for example, from > to >=, or the operand, for example, switching local variables defined in the procedures. Non control statements are modified by changing values of local variables defined within their scope.

```
public void onCreate(Bundle savedInstanceState) {
   addButton.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
         try {
        int val1 = Integer.parseInt(value1.getText().toString());
            int val2 = Integer.parseInt(value2.getText().toString());

            Integer answer = val1 + val2;
            result.setText(answer.toString());

         } catch (Exception e) {
            Log.e(LOG_TAG, "Failed to add numbers", e);
         }
      }
   });
}
```

Figure 11: Example of AIC

5.2 Variables and Measures

5.2.1 Independent Variables

The independent variables that we used in the empirical study are the code instrumentation framework and the regression test selection for Android applications. To study RQ1 and RQ2, we use CCG, which involves the *instrumentCCG* algorithm

described in Section 4.3, to compare with Emma (http://emma.sourceforge.net/), a built-in testing framework for Android Platform. To study RQ3 and RQ4, we compare the efficiency and the effectiveness of TSFA with RAS.

5.2.2 Dependent Variable and Measures

To study our research questions, we selected three dependent variables and measures which are relatively used to determine the costs of CCG versus Emma, and TSFA versus RAS. Given an original program *P,* which is modified to a new version *P'*. Note that the costs can be measured differently depending on what technique is being applied. CCG and Emma are only applied to *P*. For TSFA, only CCG is applied to *P*, and IA and TCS are applied to *P'*.

The first dependent variable is *execution time.* To study RQ1, we need to measure execution time to compare the efficiency of CGG versus Emma. We denote T_{CCG} and T_{Emma} for the execution time of CGG and Emma, respectively. Since execution of CGG includes execution of instrumentation code phase and log file analysis, we denote them as T_{IC} and T_{LA}, respectively. To study RQ4, we also need to measure execution time to compare the efficiency of TSFA and RAS. The execution time of TSFA is also divided into the execution time of its components, namely, IA and TCS. We denote T_{IA} and T_{TCS} for the execution time of IA and TCS, respectively. We also denote T_{full}, and T_{sub} as the execution time of the test suite and its subset, respectively.

The second dependent variable is *code coverage*. To study RQ2, although execution time determines the efficiency, code coverage also needs to be measured to guarantee the correctness since TSFA is strictly based on it. We denote C_{CCG} and C_{Emma} as code coverage for CGG and Emma, respectively.

The third dependent variable is *the number of selected test cases*. To study RQ3, we need to measure the number of selected test cases since we are reducing the number of test cases. In addition, not all test cases are the same in terms of length and testing criteria. For example, some test cases are short in length, but due to the testing criteria, they are executed by a large number of blocks of code. However, there are also some long test cases that are only executed by a small number of blocks of code. Hence, reduction in the number of test cases does not necessarily correlate with reduction in time. Therefore, we need to take the number of selected test cases into consideration. We denote N_{sub} as the number of selected test cases.

5.3 Experiment Setup

We run TSFA by using the Android Development Environment, together with the Ant tool and Eclipse. We use the Windows operation system running 3.4GHz with 8GByte of memory to conduct our study. We automate the process from analyzing change impact, generating code coverage reports, to selecting test cases by running batch file under Windows operation system.

For each artifact:

- We applied CCG and Emma to *P* and collected T_{CCG}, T_{Emma}, C_{CCG}, C_{Emma}, T_{full}, T_{IC}, and T_{LA}

For each version of *P:*

- We applied TSFA and collected T_{IC}, and T_{LA}

For each version of *P'*:

- We applied RAS and collected T_{RAS}.
- We applied TSFA and collected T_{LA}, T_{TCS}, T_{sub}, and N_{sub}

5.4 Threats to Validity

The primary threats to external validity for our study are (1) the changes applied to the modified programs, (2) selection of artifacts used to evaluate TSFA. The actual changes might or might not emulate the changes that we applied to the modified program. However, to control this threat, we systematically vary the changes based on their location, change type, and number of changes. Although only two artifacts are selected to evaluate our approach, they consist of all components of the Android Development Architecture, which is relative to most of popular large-scale Android applications.

The primary threats to internal validity are possible faults in the implementation of algorithms, and in tools that we use to perform the evaluation. We controlled this threat by using small programs that cover all scenarios. For such programs, we can manually generate the expected results and compare with the actual results.

With respect to threats to construct validity, the metrics that we used to evaluate the cost of TSFA are commonly used to measure the cost of regression test selection technique.

5.5 Results and Analysis

In this section, we present the results of our experiments, and analyze the results with respect to our four research questions.

RQ1. How efficient is our Code Coverage Generator compared to an existing Code Coverage Generator Emma?

Table 2 presents the results of execution time for CGG and Emma. In the table, for each artifact, the results are presented for our two dependent variables, namely, time and code coverage. To present the results for code coverage, we divided it into two

different components, namely, “executed blocked” and “ratio”. Thus, columns with header “Executed Block” represent the number of blocks that are executed by the test suite. Columns with header “Ratio” represent the percentages of blocks that are executed. To present the execution time, we accumulate T_{IC}, T_{full}, and T_{LA} into columns with header “Total”.

Table 2. Results of execution time for CCG and Emma

Apps	Number of test cases	T_{CCG}				T_{Emma}			
		T_{IC} (s)	T_{full} (s)	T_{LA} (s)	Total (s)	T_{IC} (s)	T_{full} (s)	T_{LA} (s)	Total (s)
Inetify	206	1	134	1	**136**	1	134	206	**341**
AndStatus	99	1	495	1	**497**	1	495	99	**595**

We observed that, for each artifact, T_{IC}, and T_{full} for both CGG and Emma are equal, though T_{LA} remains different. It only takes one second for CGG to analyze one single log file and generates code coverage report for each test case. Note that, by default, Emma generates one single code coverage report for the entire test suite. However, this is not applicable for regression testing where each individual test case coverage is required. We need to reconfigure Emma to fix this issue by repeating the process of executing a test case, and analyzing its log file. That is, for Emma, it takes one second to analyze a log file. Since there are 206 test cases, it would take 206 seconds to analyze the log files. Hence, the time that it takes for Emma to generate code coverage depends on both test suite execution time and number of test cases. Our CCG, as opposing, only depends on test suite execution time. We conclude that, for the use of regression testing, CCG performs more efficient than Emma.

RQ2. Does the Code Coverage Generator achieve the same level of correctness

compared to Emma? In Table 3, we measure the number of generated blocks and the number of executed blocks in order to calculate code coverage for each artifact. Note that CGG generates 71304 blocks and Emma generates 65904 blocks.

Table 3. Results of code coverage for CCG and Emma

Apps	Number of Test cases	CCG			EMMA		
		Generated Blocks	Executed Blocks	$C_{CCG}(\%)$	Generated Blocks	Executed Blocks	$C_{Emma}(\%)$
Inetify	206	7403	4930	**66.6**	6691	4452	**66.5**
AndStatus	99	71304	41071	**57.6**	65904	37770	**57.3**

During the process of code instrumentation, we insert some virtual blocks such as entry blocks and ending blocks for each CFG, which causes the greatest number of blocks. In addition, since we do not have knowledge of Emma's implementation, we are unable to verify what type of measurement being used to count number of blocks causing the less number of generated blocks. Despite the difference in the number of blocks, CCG and Emma achieve almost the same in precision for code coverage.

Table 3 and Table 4 present the results of applying TSFA and RAS to our artifacts. For the original program, columns with headers T_{RAS} , T_{full} , T_{IC}, T_{LA} represent the execution time of RAS, test suite, instrumentation code and log analysis, respectively. For the modified program, columns with headers T_{TSFA}, T_{IA}, T_{TCS}, T_{sub} represent the execution time of TSFA, IA, TCS and a subset of the test suite, respectively. Columns with header N_{sub} represent the number of selected test cases.

RQ3. How does the number of test cases selected by TSFA compare to RAS, which re-runs all test cases? Table 4 (a-c), show the results of running TSFA and RAS on each version of the artifact *AndStatus*. Note that, in Table 4b, N_{sub} for $v0 \rightarrow v3$ and $v0 \rightarrow v4$ is

an interesting observation. Even though only three changes are made at different locations of the program, the difference in numbers of selected test cases for the two versions is 28 (46 - 18). We calculate the average of numbers of selected test cases for one, three, and five changes with the result of 37 test cases. The average of 38% of the test suite is reduced for re-execution. Overall, the result meets our expectation. Compare to RAS, our approach effectively reduces the number of test cases.

Table 4 (a-c). Results of Applying TSFA and RAS to AndStatus app

Table 4a. Five versions of one change

One Change	Original Program			Modified Program					
	Our Approach			RAS (s)		Our Approach			
	Instrumentation Code (s)	Full Test Suite (s)	Log Analysis (s)	#of test cases	RAS (s)	Impact Analysis (s)	Select TC (s)	Subset of TCs (s)	# of selected test
v0→v1	2	495	1	99	495	2	1	40	**8**
v0→v2	2	495	1	99	495	2	1	45	**9**
v0→v3	2	495	1	99	495	2	1	20	**4**
v0→v4	2	495	1	99	495	2	1	50	**10**
v0→v5	2	495	1	99	495	2	1	40	**8**

Table 4b. Five versions of three changes

Three Changes	Original Program			Modified Program					
	Our Approach			RAS (s)		Our Approach			
	Instrumentation Code (s)	Full Test Suite (s)	Log Analysis (s)	#of test cases	RAS (s)	Impact Analysis (s)	Select TC (s)	Subset of TCs (s)	# of selected test
v0→v1	2	495	1	99	495	2	1	200	**40**
v0→v2	2	495	1	99	495	2	1	185	**37**
v0→v3	2	495	1	99	495	2	1	90	**18**
v0→v4	2	495	1	99	495	2	1	230	**46**
v0→v5	2	495	1	99	495	2	1	220	**44**

Table 4c. Five versions of five changes

Five Changes	Original Program			Modified Program					
	Our Approach			RAS (s)		Our Approach			
	Instrumentation Code (s)	Full Test Suite (s)	Log Analysis (s)	#of test cases	RAS (s)	Impact Analysis (s)	Select TC (s)	Subset of TCs (s)	# of selected test
v0→v1	2	495	1	99	495	2	1	275	**55**
v0→v2	2	495	1	99	495	2	1	335	**67**
v0→v3	2	495	1	99	495	2	1	320	**64**
v0→v4	2	495	1	99	495	2	1	350	**70**
v0→v5	2	495	1	99	495	2	1	330	**66**

RQ4. How efficient is the time reduction when applying TSFA compared to RAS?

Table 5 (a-c) shows the results of running TSFA and RAS on each version of the artifact *Ientify*. Each table presents results of one, three and five number of changes, together with five versions of each. For example, Table 4a can be interpreted as the following:

Given an original program P, and its modified program P', the number of changes between P and P' is one. For each version from $v0 \rightarrow v1$ to $v0 \rightarrow v5$, we

calculate T_{RAS} and T_{TSFA} for P'. For example, with v0 → v1, the overhead is small because it only takes 2 seconds to perform instrumentation code and log analysis. With N_{sub} = 56 in P', it takes 39 seconds ($T_{IA}+T_{TCS}+T_{sub}$) to run. Hence, it takes TSFA the total of 41 seconds on *P'*. For RAS, with v0 → v1, it takes 134 seconds for RAS to run on P'.

When there is one change in the application, Table 5a shows that it takes 41 seconds for TSFA to run and 134 seconds for RAS to run on *P'*. So the execution time is reduced by 69 %. We calculate the average for the time reduction for one, three, and five changes with the result of 75%. We observed that as long as the number of changes stays low, the execution time is substantially reduced. Overall, the execution time is efficiently reduced when applying TSFA compared to RAS.

Table 5 (a-c). Results of Applying TSFA and RAS to Inetify app

Table 5a. Five versions of one change

One Change	Original Program			Modified Program					
	Our Approach			RAS (s)		Our Approach			
	Instrumentation Code (s)	Full Test Suite (s)	Log Analysis (s)	#of test cases	RAS (s)	Impact Analysis (s)	Select TC (s)	Subset of TCs (s)	# of selected test
v0→v1	1	134	1	206	134	1	1	37	**56**
v0→v2	1	134	1	206	134	1	1	25	**38**
v0→v3	1	134	1	206	134	1	1	26	**39**
v0→v4	1	134	1	206	134	1	1	24	**36**
v0→v5	1	134	1	206	134	1	1	35	**53**

Table 5b. Five versions of three changes

Three Changes	Original Program			Modified Program					
	Our Approach			RAS (s)		Our Approach			
	Instrumentation Code (s)	Full Test Suite (s)	Log Analysis (s)	#of test cases	RAS (s)	Impact Analysis (s)	Select TC (s)	Subset of TCs (s)	# of selected test
v0→v1	1	134	1	206	134	1	1	71	**108**
v0→v2	1	134	1	206	134	1	1	41	**63**
v0→v3	1	134	1	206	134	1	1	62	**95**
v0→v4	1	134	1	206	134	1	1	73	**111**
v0→v5	1	134	1	206	134	1	1	56	**86**

Table 5c. Five versions of five changes

Five Changes	Original Program			Modified Program					
	Our Approach			RAS (s)		Our Approach			
	Instrumentation Code (s)	Full Test Suite (s)	Log Analysis (s)	#of test cases	RAS (s)	Impact Analysis (s)	Select TC (s)	Subset of TCs (s)	# of selected test
v0→v1	1	134	1	206	134	1	1	89	**136**
v0→v2	1	134	1	206	134	1	1	91	**140**
v0→v3	1	134	1	206	134	1	1	82	**126**
v0→v4	1	134	1	206	134	1	1	106	**162**
v0→v5	1	134	1	206	134	1	1	112	**171**

6. DISCUSSION

Table 6 (a-b) presents the results of total execution time for each artifact. For example, when the number of changes is one, RAS takes 134 seconds to complete regression testing of v1 of the *Inetify* application. TSFA takes 41 seconds. We also use average ratios of total time of TSFA and total time of RAS to express the efficiency and effectiveness of TSFA on Graph 1 and Graph 2.

Table 6 (a-b) shows the consistency of the results when applying TSFA. Our study focuses on reducing the number of test cases after the original program is modified. According to Graph 1 and Graph 2, as the number of changes increases, the total execution time also increases since the number selected test cases increases. This observation meets our expectation.

The number of selected test cases can be affected by testing criteria. For instance, if the testing criteria are focusing on one particular part of an application, then the majority of the test cases will be executed by that part. On the one hand, one change from that part can select the majority of test cases. On the other hand, many changes from different parts of the application can only select a small number of test cases. To resolve this issue, we have chosen a systematic way to create versions of modified programs with changes that vary in locations, type change, and the number of changes.

Different from the *Inetify* app, the *AndStatus* app is a much larger scale Android application based on the information from Table 1. Note that, the number of test cases executed on the *AndStatus* app (99) is only a half of the number of test cases executed on the *Inetify* app (206). Based on these two facts, we can predict that the test suite for the AndStatus app will result in low code coverage since there is not enough test cases to

cover the large system. Hence, the number of the selected test cases will decrease. Therefore, we can predict that only a small number of test cases are selected for the *AndStatus* app, which reduces the execution time. However, Table 6b shows that, after one change, the average of 54% of time reduction remains. This is an interesting observation. Despite our prediction, the execution time does not remain low. The factor that makes our prediction incorrect is the characteristic of the test suite. Without examining the test suite, we can conclude that the majority of the test cases is greater in length. Moreover, the test criteria upon which they are designed covers a large number of blocks in the application.

We assume that the characteristic of the test suite results in low execution time. Then we would have correctly predicted the outcome. This is when the trade-off between the time required to run selected test cases and fault detection ability comes in. Suppose that many changes have found in the program. The greater number of test cases is eliminated, the less effective is the ability to detect faults. Therefore, three factors that can affect our study are the AUTs, the characteristic of the test suite, and the program changes.

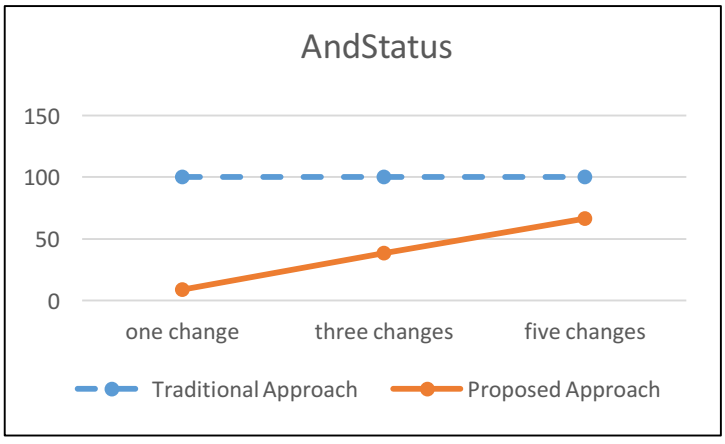
Table 6 (a-b). Total Execution Time of TSFA and RAS

Table 6a. Inetify Application

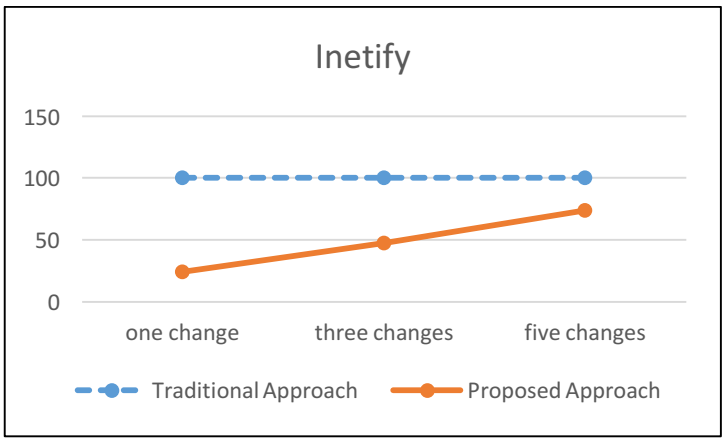
Version	1 change			3 change			5 change		
	T_{RAS}(s)	T_{TSFA}(s)	Ratio	T_{RAS}(s)	T_{TSFA}(s)	Ratio	T_{RAS}(s)	T_{TSFA}(s)	Ratio
$v0 \rightarrow v1$	134	41	0.31	134	75	0.56	134	93	0.7
$v0 \rightarrow v2$	134	29	0.22	134	45	0.34	134	95	0.71
$v0 \rightarrow v3$	134	30	0.23	134	66	0.5	134	86	0.65
$v0 \rightarrow v4$	134	28	0.21	134	77	0.58	134	110	0.83
$v0 \rightarrow v5$	134	39	0.3	134	60	0.45	134	116	0.87

Table 6b. AndStatus Application

Version	1 change			3 change			5 change		
	T_{RAS}(s)	T_{TSFA}(s)	Ratio	T_{RAS}(s)	T_{TSFA}(s)	Ratio	T_{RAS}(s)	T_{TSFA}(s)	Ratio
$v0 \rightarrow v1$	495	46	0.1	495	206	0.42	495	281	0.57
$v0 \rightarrow v2$	495	51	0.11	495	191	0.39	495	341	0.69
$v0 \rightarrow v3$	495	26	0.06	495	96	0.2	495	326	0.66
$v0 \rightarrow v4$	495	56	0.12	495	236	0.48	495	356	0.72
$v0 \rightarrow v5$	495	46	0.1	495	226	0.46	495	336	0.68



Graph 1: Total Execution Time of the AndStatus Application



Graph 2: Total Execution Time of the Inetify Application

7. RELATED WORK

Many researches have been conducted only focusing on testing a single version of applications. For example, Azim T., Neamtiu I. (2013) presented an approach that allows popular Android apps to be explored systematically while running on an actual phone. This work successfully addressed its purpose in term of GUI exploration. Since the work is done without having access to source code, but by the analysis of bytecode, all possible transitions (i.e. method invocations) to new activities should be taken into account. Another single version application testing example is the work of Yang W., Prasad M. R., and Xie T. (2013). The researchers proposed a tool that implements the grey-box approach of automated model extraction for Android apps and its evaluation in demonstrating the effectiveness at generating high-quality GUI models.

Domenico A., Fasolino A. R., and Tramontana P. (2011) proposed a type of regression testing of Android applications using the Monkey Runner tool to check the modifications by comparing the output screenshots to a set of screenshots that are known to be correct. This approach only provides a high level of comparability to find the changes in the modified versions.

Another research conducted by Crussell J., Clint G., and Hao C. (2012) detects the similarity between an Android application's versions by constructing Program Dependency Graph (PDG) for each version using an existing tool and comparing the PDGs to find the semantical difference of code at the method level. Compared to our approach, we construct CFGs (similar to PDGs) from scratch, instead of using an existing tool. This provides a more flexible way to build and compare the CFGs. Then we detect program changes at the syntax level, not semantic level.

Memon A. M., and Soffa M. L. (2003) proposed a regression testing technique for GUIs application in general. The study only focuses on regression testing of desktop applications.

8. CONCLUSION AND FUTURE WORK

We have presented a new approach for regression test selection of mobile platform. Our approach leverages the concept of static analysis to detect the modifications on different versions of an Android application. Because of the changes, only a subset of test cases is selected to avoid the cost of executing the entire test suite. More importantly, our study has fulfilled the gap of lacking of regression test selection technique on Android applications. We also evaluate the efficiency and the effectiveness of our approach using an empirical study.

As far as the availability of code coverage tools, they are mainly used to collect code coverage information of programs. However, none of them is specifically developed for regression testing of Android applications. CCG has the ability to generate code coverage for each individual test case in an efficient way, whereas other tools can only generate code coverage for the entire test suite, which cannot be beneficial in selecting test cases.

One limitation of our approach is that we only leverage the concept of static analysis at source code level. Changes in different places such as XML layout files, library, hardware, etc. In the future, we plan to optimize a broader range of static analysis that is used for the impact analysis in our framework. In addition, not only do we focus on the physical changes of the source code, but also the changes in behavior of Android applications.

Moreover, we want to conduct more evaluation for our study. Instead of using two artifacts, we will evaluate our approach on additional Android applications. Since the

changes in the modified versions of the programs are randomly seeded, we will improve the evaluation by using changes made by real developers.

APPENDIX

instrumentCGG{

```
for (ih = il.getStart(); ih != null; ih = ih.getNext()) {
Iterator lead = leaders.iterator();
while (lead.hasNext()){
  if(ih.equals(lead.next())){

  setInstrument(il,cpg,cg,INSTRUMENTSTRING,"~"+cg.getClassName()+","+mg.getName()+","+ih.getPo
sition()+": "+ih.getInstruction().toString() ,ih);
      numInstru++;
      if(!(ih.getInstruction() instanceof ReturnInstruction) )
          leaderMap.put(ih, moveToNewLeader(ih));
      else
          leaderMap.put(ih, ih);
  }
}
}

/*
* Update position
*/
il.setPositions();

/*
* Update targetMap
*/
Iterator it = targetMap.entrySet().iterator();
while (it.hasNext()) {
Map.Entry pairTarget = (Map.Entry)it.next();
Iterator it2 = leaderMap.entrySet().iterator();
while (it2.hasNext()) {
  Map.Entry pairLeader = (Map.Entry)it2.next();
  if(pairTarget.getValue().equals(pairLeader.getKey())){
      pairTarget.setValue(pairLeader.getValue());
  }
}
}

/*
* Update targets for new bytecode
*/
for (ih = il.getStart(); ih != null; ih = ih.getNext()) {
Iterator it3 = targetMap.entrySet().iterator();
while (it3.hasNext()) {
  Map.Entry pairLeader1 = (Map.Entry)it3.next();
  if(ih.equals(pairLeader1.getKey())){
      insn = ih.getInstruction();
      if (insn instanceof Select){
          InstructionHandle ih2 = (InstructionHandle)pairLeader1.getValue();
          Select selectInstr = (Select) insn;
          //Defaut case
          target = selectInstr.getTarget();
          if(leaderMap.containsKey(target)){
              InstructionHandle newTarget = (InstructionHandle) leaderMap.get(target);
              selectInstr.updateTarget(target, newTarget);
          }
          // Case Targets
          InstructionHandle [] targets = selectInstr.getTargets();
          for(int k = 0; k < targets.length;k++){
```

```
                        if(leaderMap.containsKey(targets[k])){
                            InstructionHandle newTarget = (InstructionHandle) leaderMap.get(targets[k]);

                            selectInstr.updateTarget(targets[k], newTarget);
                        }
                    }
                }else{
                    InstructionHandle ih2 = (InstructionHandle)pairLeader1.getValue();
                    ((BranchInstruction) insn).setTarget(ih2);
                }

            }
        }
        }

        /*
        * update new leaders
        */
        leaders.clear();
        Iterator it4 = leaderMap.entrySet().iterator();
        while (it4.hasNext()) {
        Map.Entry pairLeader1 = (Map.Entry)it4.next();
        leaders.add(pairLeader1.getValue());
        }
        }

        // Add calculated basic blocks
        Iterator leadersIt = leaders.iterator();
        Iterator endIt = ends.iterator();
        while (leadersIt.hasNext() && endIt.hasNext()) {
        InstructionHandle startHandle = (InstructionHandle) leadersIt
          .next();
        InstructionHandle endHandle = (InstructionHandle) endIt.next();

        Node block = new Node(startHandle.getPosition(),
          endHandle.getPosition());
        block.setStartHandle(startHandle);
        block.setEndHandle(endHandle);
        allBasicBlocks.put(startHandle.getPosition(), block);
        }
}
```

REFERENCES CITED

Android Developer Guide. *Android Developers*. N.p., n.d. Web. 20 Oct. 2015. http://developer.android.com/

Azim T., and Neamtiu I. (2013). Targeted and Depth-first Exploration for Systematic Testing of Android Apps. *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. Pages 641-660.*

Chen Y., D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. *In Proceedings of the 16th International Conference on Software Engineering, pages 211-222.*

Crussell J., Clint G., and Hao C. (2012). Attack of the Clones: Detecting Cloned Applications on Android Markets. *17th European Symposium on Research in Computer Security, Pisa, Italy.*

Domenico A., Fasolino A. R., and Tramontana P. (2011). A GUI Crawling-based technique for Android Mobile Application Testing. I *CSTW '11 Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. Pages 252-261*.

EMMA: A free Java code coverage tool (n.d). Retrieved October 20, 2015 http://emma.sourceforge.net/

Fischer K., Raji F., and Chruscicki A. (1981). A methodology for retesting modified software. *In Proceedings of the National Telecommunications Conference B-6-3, pages 1-6.*

Harrold M. J., and Rothermel G. (1997). A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Meth., 6(2):173–210.*

Hartmann J. and D. Robson. Techniques for selective revalidation. IEEE Software, 16(1):31- 38.

Hiralal A., Joseph R. H., Edward W. K., Saul A. L. (1993). Incremental Regression Testing. *IEEE Conference of Software Maintenance.*

Inetify. (n.d). Retrieved October 20, 2015. https://code.google.com/p/inetify

Memon M. Memon, Soffa Mary Lou. Regression Testing of GUIs. ESEC/FSE-11 *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering. Pages 118-127.*

Monkeyrunner .(n.d). Retrieved October 20, 2015 http://developer.android.com/tools/help/monkeyrunner_concepts.html

Rothermel G. and Harrold M. J. (1996). Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng., 22(8):529–55.*

Rothermel G. and Harrold M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology, 6(2):173-210.*

Todd L. G., Mary J. H., Jung-Min K., Adam P., Gregg R. (2001). An Empirical Study of Regression Test Selection Techniques, *ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2.*

Volkov, Y. (n.d). AndStatus. Retrieved October 20, 2015 https://github.com/andstatus/andstatus

Yang W., Prasad M. R., and Xie T. (2013). A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. *FASE'13 Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering. Pages 250-265*.