ReTestDroid: Towards Safer Regression Test Selection for Android Application*

Bo Jiang, Yu Wu, Yongfei Zhang
School of Computer Science and Engineering
Beihang University
Beijing, China
{jiangbo,yuwu,yfzhang}@buaa.edu.cn

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software,
Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

W.K. Chan†
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

***Abstract*—Mobile applications are widely used in our daily life and Android is the most popular open source mobile operating system. Because mobile applications update frequently, it is important developers to perform regression testing to ensure their quality. Modeling the control flow of an android application based on the activity lifecycle model only is imprecise for regression testing. Because many Android applications use asynchronous tasks, fragments, and native code frequently, which must be considered during change impact analysis. Otherwise, regression test selection techniques may miss some failure-revealing test cases, compromising the safety of these techniques. In this work, we propose a novel approach to model asynchronous task invocations, fragment-based activity lifecycle, and native code within the control flow graph of an Android application. Furthermore, we designed a regression test selection tool ReTestDroid based on our graph model. Our experiments on five real-life Android applications showed that our approach could enable much safer regression test selection while significantly saving regression-testing time.**

Keywords—Test Case Selection; Android; Regression Testing; Impact Analysis

I. INTRODUCTION

With the prevalence of smart phones and mobile operating system (e.g., iOS & Android), mobile applications are becoming an indispensable part of our life. From the point of view of the mobile application developers, those mobile applications serve as a crucial interface of their business services to end-users. Many popular mobile applications (e.g., Facebook and Wechat) have hundreds of millions of active end users, which is the key to the business success of the company. Indeed, a low quality mobile application will seriously impact user experiences. Thus, mobile developers strive to ensure the quality of their mobile applications to avoid user loss.

A key characteristic of such a mobile application is that their software components undergo rapid evolutions [2]. In another word, newer versions of the same mobile application are released frequently. For instance, Firefox is planned with tens of official releases (versions 45.8 to 52.7) and another tens of developer releases (versions 53.0 to 62.0) in 2017. Furthermore, a survey on Android Play store [41] reports that such a period is around 10 days for apps with more than 100K+ downloads. Therefore, in such a short period, not only the source code of an app is modified but also all the testing should be completed toward the release of the new version.

Regression testing is the activity of testing changed software to provide confidence that the changed parts of the software behaves as expected and that the unchanged parts of the software have not been adversely affected [23]. There are many regression-testing techniques studied in the literature. One important technique is Regression Test Selection (RTS), which selects a subset of test cases (denoted as test suite A) for regression testing, rather than re-testing all these test cases (denoted as test suite B), on a newer version [29][32] based on some notions of equivalence. For instance, if test cases in A and B both pass through the same set of edges in the same control flow graph of a version of an app (called the original version in RTS), then test suite A may be selected to test a new version of the app for the regression testing. Furthermore, if the control flow graphs of the two app versions are available, then the set of nodes (i.e., program statements) and the edges of the control flow graph of the original version can be labeled to indicate that these nodes and edges are impacted by changes between the two control flow graphs. The test suite A can be further reduced to merely include test cases that pass through these edges impacted by change. This process is known as change impact analysis.

To realize such change impact analyses, a key issue is to construct a control flow graph that precisely models the application. For Android applications, the state of the art technique [2] is to model the control flow based on the source code while also incorporating the activity lifecycle model. Each lifecycle model is a graph where a node represents an activity state and an edge represents a transition between two activity states. Nonetheless, many Android applications uses fragment class to build modularized user interfaces. A fragment represents a behavior or a portion of user interface in an Activity. When fragment class is used, the event handlers of

* This research is supported in part by the National Natural Science Foundation of China (project no. 61772056), the Key Research Fund of the MIIT of China (project no. MJ-Y-2012-07), the General Research Fund of Research Grants Council of Hong Kong SAR (project numbers: 125113, 11200015, 11201114, and 11214116), and the research funding of the State Key Laboratory of Virtual Reality Technology and Systems.
† Correspondence author

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00037

the fragment class are used instead of event handlers of the Activity class. However, the ICFGs built on top of the Activity lifecycle have no place to accommodate fragment event handlers. When changes happen in those fragment event handlers, using such a less precise graph may lead to the omission of failure-revealing test cases, which is undesirable. Furthermore, the asynchronous task is frequently used on Android for parallel processing and native code is used in Android for improving efficiency. These two programming features are also not reflected in the control flow graph of existing work [2].

Although there are many existing works on RTS [4][9][15][18][20][27][32][35][36], they exclusively focus on individual programming languages (e.g., Java) and individual programming models. Nonetheless, for Android apps, we argue that effective change impact analysis for RTS requires handling asynchronous tasks, fragments, and native code appropriately, which we will address in this work.

In this paper, we propose a novel Inter-Procedural Control Flow Graph (ICFG) to support regression testing of Android-application with asynchronous tasks, fragment-based activity lifecycles as well as native code. Our approach not only models transitions at the activity fragment levels, but also handles native code and asynchronous task invocations. As such, our graph model is more precise in detecting code changes between versions of the same application. Consequently, with our control flow abstraction, change impact analysis for regression testing is also more precise. To show the feasibility of our approach, we have implemented it as a Regression Test selection system for AnDroid applications (ReTestDroid). We have conducted experimentation to evaluate ReTestDroid in regression testing scenarios on five real-world applications. The results show that the interprocedural control flow graph proposed by ReTestDroid is effective to support safer regression test selection. Furthermore, ReTestDroid can significantly reduce the test suite size as well as the overall regression testing time, which is promising for practical use.

The contribution of our work is twofold. It is the first work to propose precise Interprocedural Control Flow Graph (ICFG) for Android application by handling asynchronous task, fragment, and native code in supporting effective RTS. Second, this paper presents the first experimental study on large real-life Android applications (e.g., Mozilla FireFox Mobile, K9-Mail) to evaluate the safety of the regression test selection system.

The rest of this paper is organized as follows. Section II presents the background relevant to our work. In Section III, we present our regression test selection system ReTestDroid, which includes the overall system workflow, the ICFG construction for Android apps, and the impact analysis algorithms. Section IV provides our experimental study on real-life Android apps. Finally, Section V reviews related work and Section VI concludes the paper.

II. BACKGROUND

In this section, we present the preliminaries and the background on graph representations of programs for control flow analysis as well as the regression test selection problem.

A. Graph Representations of Programs for Control Flow Analysis

A Control Flow Graph (CFG) for a method *M* contains a node for each simple or conditional statement in *M*. The edges between nodes represent the flow of control between statements. Usually, intraprocedural control flow analysis is performed on one method at a time. When connecting the static call graph with the control flow graphs of all the procedures in an application *P*, we can build an *Interprocedural Control Flow Graph* (*ICFG*) [42] of the whole application. Specifically, an *ICFG* for a program *P* is composed of the *CFGs* for each method in *P*. Each call site in *P* is represented by a pair of call and return nodes. The call node is connected to the entry node of the called method by a call edge, and each exit node in the called method is connected to the return node of the calling method by a return edge.

For applications written in Object-Oriented (OO) programming languages such as Java or C++, existing works extended the ICFG to the Java Interclass Graph (JIG) [32] or Class Control Flow Graph (CCFG) [36] to handle the OO features such as inheritance, polymorphism as well as framework programming features. The construction of JIG enables a more precise control flow analysis on the whole Java application for regression testing purpose, which is significant.

FlowDroid [2] provides a solution for constructing Inter-Procedural Control Flow Graph (ICFG) of Android apps. It then uses its ICFG to perform static taint analyses and applies it for detecting leaks of sensitive user data. The ICFG built by FlowDroid accommodates the activity lifecycle of Android apps, inserts the events handling callbacks defined by the Android framework, connects multiple entry points of Android components (Activity, Service, Broadcast Receiver, and Content Provider) within a dummy main function, and handles the Object-Oriented features of Android programming such as inheritance and polymorphism.

However, as mentioned in Section 1, the state-of-the-art ICFG built by FlowDroid is limited in at least three aspects necessary for safer regression testing with change impact analysis: It does not handle the call to Android framework APIs related to asynchronous tasks, it does not model Fragment lifecycle, and it does not handle native code. In this work, we address these three limitations in our ICFG construction process so that the ICFG can be precise for change impact analysis from the control flow perspective.

B. The Safe Regression Test Selection Problem

In regression testing research, the retest-all strategy [24] is to execute all the test cases in an existing regression test suite over the modified software. Regression Test Selection (*RTS*) is to select a subset of test cases from a given test suite (*T*). The regression test selection essentially consists of two major activities [7]:

1) Impact Analysis: Identification of the unmodified parts of the program that are affected by the modifications.
2) Test Case Selection: Identification of a subset of test cases from the initial test suite T which can effectively

test the affected parts identified by the previous activity.

Rothermel and Harrold [32] formally defined the regression test selection problem as follows: Let *P* be an application program and *P'* be a modified version of *P*. Let *T* be the test suite developed initially for testing *P*. *RTS* technique selects a subset of test cases *T'* of *T* to be executed on *P'*, such that every error detected when *P'* is executed with *T* is also detected when P' is executed with *T'*.

A test case *t* of *T* is considered modification-revealing [33] for *P* and *P'*, if and only if it produces different outputs for *P* and *P'*. A test case *t* of *T* is said to be modification-traversing for *P* and *P'* if and only if the execution traces of *t* on *P* and *P'* are different. If traces are the same, then the outputs will be the same. They define a test case selection algorithm as safe if it selects all test cases that are modification revealing. In this work, we consider a test case selection technique **safe** if all failure-revealing test cases have been selected for a program version. Furthermore, we consider one RTS technique is **safer** than the other if the former can select more failure-revealing test cases than the later on a program version.

III. RETESTDROID: OUR REGRESSION TEST SELECTION SYSTEM

The regression test case selection strategy stated at the end of last section implies that if an ICFG used for RTS omits some essential nodes or edges that reflect the program control flow, regression test selection cannot be safe. As we have discussed in *Section I*, it is exactly the case on the ICFG generated by FlowDroid. In this section, we present our regression test selection system ReTestDroid for Android apps. We first present the overall workflow of our ReTestDroid framework, and then describe how we build a more precise ICFG for Android app to address the three limitations stated in the last section. After that, we show the regression test selection algorithms. Finally, we discuss the limitations of ReTestDroid.

A. Workflow of ReTestDroid

In this section, we present the workflow of our regression test selection system (ReTestDroid), which is shown in Fig. 1.

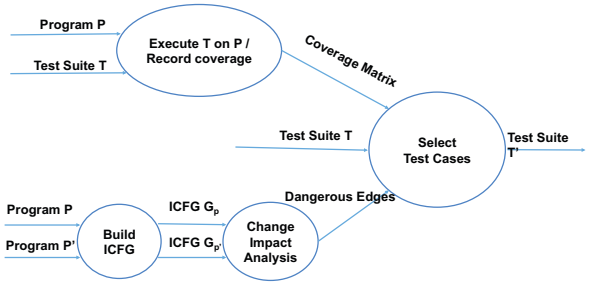


Fig. 1 The Workflow of ReTestDroid

Given an Android app with two versions *P* and *P'* as well as a test suite *T*, ReTestDroid statically build two ICFGs G_p and G_p' for *P* and *P'*, respectively. After that, ReTestDroid performs change impact analysis on G_p and G_p' to label a set of edges on G_p as dangerous. Then, ReTestDroid executes *P* over *T* to generate the coverage matrix of *T* on *P* with respect to G_p to indicate which edges in G_p have been exercised by which test cases in *T*. This coverage information is usually collected after the testing of previous program versions in practice. Finally, ReTestDroid selects a subset *T'* from *T* based on the coverage matrix and the labeled dangerous edges. Note each edge in the *ICFG* of *P* that is modified in *P'* is called a ***dangerous*** edge.

B. Construction of the Interprocedural Control Flow Graph (ICFG) for Android Apps

The ICFG constructed by ReTestDroid significantly enhances the ICFG built by FlowDroid [1]. As discussed in previous sections, an ICFG built by FlowDroid includes the modeling of component lifecycles, callback edges, multiple entry points, as well as the Object-Oriented features of the Android apps under analysis. The ICFG of FlowDroid provides a solid yet basic framework for static analysis. ReTestDroid further enhances the ICFG built by FlowDroid with the following improvements: 1) it handles the calls to Android framework APIs related to asynchronous tasks. 2) its ICFG handles the lifecycle of Fragments; 3) it handles the native code built with *Android NDK*. In our preliminary study, these features are frequently used in Android application programming. In the next three subsections, we present how ReTestDroid achieves these improvements.

1) Handing Asynchronous Tasks

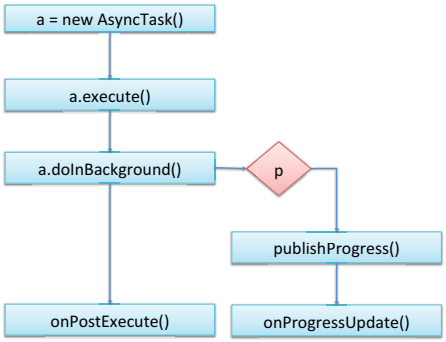
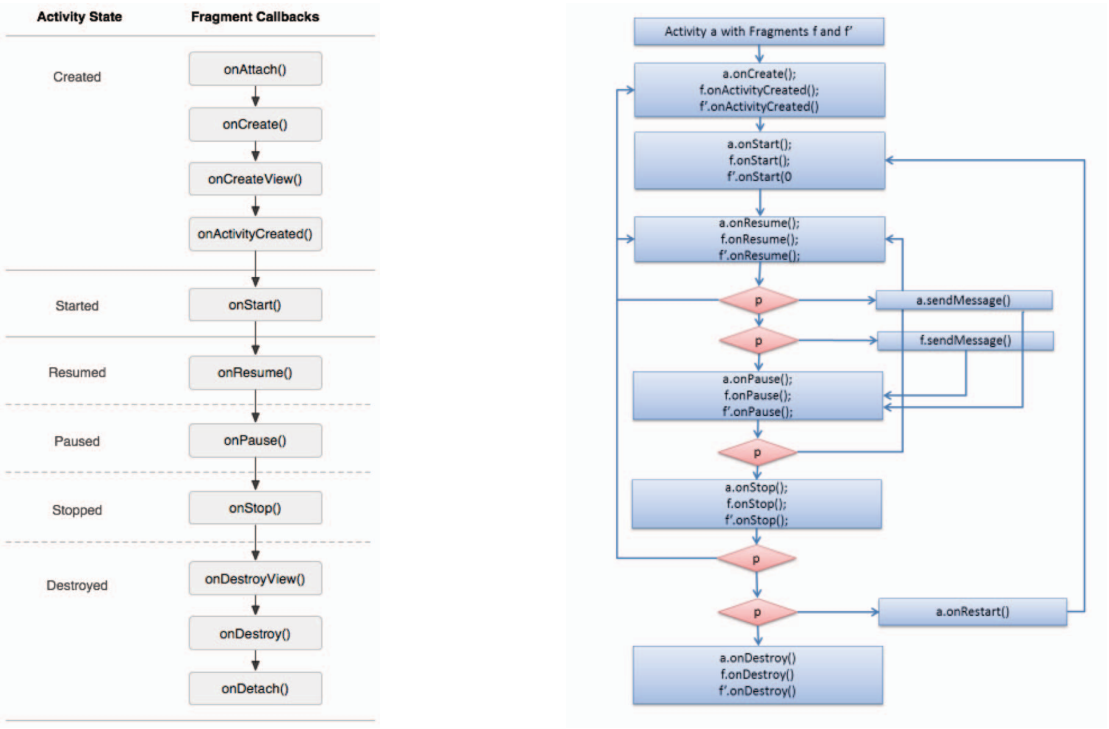


Fig. 2 Sub-ICFG Modeling Asynchronous Tasks

An asynchronous task is used by an Android app to perform background operations and publish results on the UI thread without having to use threads or handlers [45]. It is in fact the recommended way in Android for multi-threading. For asynchronous tasks, ReTestDroid connects the *execute()* method of each AsyncTask module with the *doInBackground()* callbacks implemented by that AsyncTask module. The *doInBackground()* method may optionally call the publishProgress() method, which will lead to the invocation of *onProgressUpdate()*. When *doInBackground()* returns, *onPostExecute()* will be called. ReTestDroid added all these edges within the lifecycle of *AsyncTask* in its ICFG. An exemplified sub-ICFG modeling the asynchronous tasks is



(a) (b)

Fig. 3 (a) Relationship of Fragment lifecycle and its containing Activity state. (b) Part of a sample Fragment-aware control flow graph generated by ReTestDroid.

shown in Fig. 2 where *p* is a predicate in the graph. All these sub-ICFGs are incorporated into the ICFG of ReTestDroid to enable precise impact analysis on application code with asynchronous tasks.

2) Handling Life Cycles of Fragments

In Android, a Fragment is a module of code that holds part of the behavior and/or UI of an Activity [9][13] and is subservient to an Activity. Fig. 3 (a) shows the relationship of Fragment lifecycle and its containing Activity state. Each invocation of each callback method of the containing Activity triggers an invocation of the corresponding callback method of an underlying Fragment module (e.g., onStart, onResume, onPaused, onStop, onDestroy) [13]. For instance, the onActivityCreated() callback of a Fragment module is invoked when the onCreate() method of its containing Activity module is returned. Different Fragment modules may invoke different callbacks of other Android components in different lifecycle method invocations.

The lifecycle of a Fragment is dependent on the Activity containing it. To model it, ReTestDroid inserts the call to the callbacks of Fragments right after each call to the corresponding callback of its belonging Activity. For example, when the onCreate() method of an activity is put into the ICFG, the onActivityCreated() methods of its dependent fragments will be inserted right after it. Other life-cycle methods of fragments are inserted into the ICFGs similarly.

Most importantly, those event handlers of the Fragment are also inserted into the ICFGs in between the lifecycle method of onResume() and onPause(). For example, for the class *ListFragment*, the *onListItemClick()* method will be added into the ICFGs. In contrast, in the ICFGs of FlowDroid where the Fragment class is not modeled, the fragment event handler is nowhere to go. If changes happen in those fragment related event handlers, those modification-revealing test cases covering them will not be selected since the change is not reflected anywhere in the ICFGs of FlowDroid. However, with the ICFGs of ReTestDroid, the problem is addressed.

Fig. 3 (b) shows a sub-ICFG modeling an example Activity with two Fragments generated by ReTestDroid. The *a.sendMessage()* represents arbitrary fragment related event handlers realized in the application.

3) Handing Native Code

The Android platform supports programming in native code (C and C++) using Android Native Development Kit (NDK) [44]. Precise analysis of Android apps written with native code is nontrivial. Since the SOOT framework used by FlowDroid is targeted at analyzing Java bytecode, we have to adopt a static analysis framework for C/C++ programs to perform the required analysis. To the best of our knowledge, existing analysis framework on Android application only handles the Java code and treats all calls to the native code as a system call symbol in their graph model.

TABLE 1. THE IMPACT ANALYSIS ALGORITHM

```
   Inputs: N : entry node in the ICFG for original program P
           N′: entry node in the ICFG for modified program P′
   Output E: a global set of dangerous edges for P
   methodStatus: has status ("unSelected", "selectsAll") to represent
   method impact info.
   methodTable: is a global map (methodName, methodStatus) which
   contain methods status.
   procedure compare(N, N′)
   begin
1    mark N as "N′-visited"
2    foreach(call edge or virtual edge e′ in N′.leavingEdges){
3          e = N.match(e′)  //get edge e with the same property as e'
4      m and m′ are entry nodes of the targets method of e and e′
5      if (m exist and not in methodTable)
6          compareMethod(m, m′)
       } //foreach
7    if(All target methods of N are already set "selectsAll" )
8      return  //No more analysis is need for the current method
9    foreach (normal edge e in N′.leavingEdges){
10     e = N.match(e′)              //get edge which has same property
11     if(e!=null){                 //compare target nodes of edges
12           c = e.getTarget()
13           c′ = e′.getTarget()
14           if(!e.equals(e′)){
15              E = E∪e
16           } //add edge to set E
17       else{
18          compare(c, c′)
19       } //iterate compare next nodes
20     } //end if (e′ != null)
21   } //end foreach
22   foreach(edge e in N.leavingEdges and e has no matched e′ ){
23     E = E∪e     //e can be any types edge
24   } //foreach
   end

   procedure compareMethod(N, N′)
   Input: N, N′: entry nodes of two methods
   begin
25   m is the method name for node N
26   put m in methodTable and set methodStatus ("unSelected")
27       compare(N, N′)
28       if (None of the exit nodes of m is visited)
29          set methodStatus ("selectsAll") for m
   end
```

Therefore, ReTestDroid first generates an ICFG for the native code portion. Then it connects this ICFG for the native code to the ICFG for the Java code portion to construct a combined ICFG. To generate an ICFG for the native code portion, ReTestDroid generate the call graph of the whole native code written in C/C++ as well as the intraprocedural control flow graph of each function with LLVM compiler framework. Then it connects the call graph to all the Intraprecedural CFGs of all the functions to build the ICFG of the native code portion. Finally, ReTestDroid identifies each Java Native Interface (JNI) call site in the ICFG of the Java code portion and adds an edge from that call site to the ICFG of the native code.

C. The Regression Test Selection Algorithm for Android Application

The idea of the *impact analysis algorithm* realized by ReTestDroid is an adaptation of the efficient algorithm for procedural program proposed by Rothermel et al. [34] to Obejct-Oriented program. They both try to avoid further traversal beyond a call node if all tests become modification-traversal if analyzed within the called method. However, the efficient algorithm in [34] is just for procedural programs, it takes no consideration of the *OO* features such as polymorphism. In contrast, our algorithm accommodates the virtual calls appropriately, which are frequently used in OO and Android application.

As shown in Table 1, this efficient algorithm caches the "selectsAll" tag to skip unnecessary impact analysis of follow-up nodes after a called node is analyzed. The flag "selectsAll" represents that all its successor nodes are impacted and there is no need to traverse more. Both methodStatus and methodTable are hash tables to keep the impact analysis status for a method. The efficient impact analysis algorithm starts by invoking ***compare()***, and its core idea is to handle different types of nodes in different ways (lines 2 to 6). If a node N has any call edges or virtual edges, their target node must be an entry node of a method, and so it invokes ***compareMethod()*** to perform impact analysis on that method (lines 6). If every target methods of *N* are marked as "selectsAll", then no more analysis is needed for the current method (line 7 and 8). Apart from call and virtual edges, node *N* may also have ordinary edges.

It then iterates each edge of *N′* with a matched edge of *N*, and checks whether their target nodes are equal or not (line 11 to line 14). If their target nodes match, this algorithm will recursively invoke ***compare()*** to traverse the two graphs (line 18). Otherwise, a dangerous edge is identified and added to the set *E* (initially an empty set). Finally, a loop finds whether there is any leaving edge *e* of *N* that does not have any matched edge *e′* of *N′* and adds every such edge e to set *E* (line 22 to 24). This algorithm ends after traversing the whole *ICFG* of *P*.

The method ***compareMethod***() accepts the two entry nodes of two methods, it records the method status into *methodTable* (line 26) and traverses the *CFG* by recursively invoking ***compare()*** (line 27). Only if none of the exit nodes of a method is visited, this method is set as "*selectsAll*" (lines 28 to 29).

The test case selection process is intuitive. We first recall that by executing the program *P* over the test suite *T*, the set of edges on the *ICFG* exercised by each test case is recorded, which forms a coverage matrix. As shown in Table 2, the test case selection algorithm accepts the set of dangerous edges, the coverage matrix, and the whole test suite as its inputs. It returns the set of selected test cases for *P'*. Based on the coverage matrix, it checks whether a test case covers any dangerous edges identified by an impact analysis algorithm. If this is the case, that test case is added to the set of selected test cases.

TABLE 2. THE TEST CASE SELECTION ALGORITHM

```
Input:  E: {e1, e2, ...} ← dangerous edges from impact analysis
        C: {c1, c2, ...} ← coverage matrix of the original program
        T: {t1, t2, ...} is a set of test cases for P
Output: T': {t1, t2, ...} is a set of selected test cases for P'
begin
    foreach(ci in C){
        if ( ti covers any ej in E based on ci){
            T' = T'∪ti
        } //end if
    } //foreach
end
```

D. Limitations

Our regression test selection algorithm is only safe under certain assumptions. These assumptions are also called regression bias in previous work [5]. In particular, if the application under test uses reflection mechanism or if it has non-deterministic execution behaviors, the ICFG or the test coverage matrix will change, which will makes our tool fail to be safe.

1) Reflection

Similar to the work on regression test selection on Java [29][32], the ReTestDroid framework has not supported reflection (used in any internal or external class). Similar to the Java programming environment, Android also supports reflection to access classes or their members by name. Such reflective accesses are hard to analyze [25] using static analysis: class and method names can be computed at runtime or loaded from files that the static analysis does not have access to. Since the runtime information on reflection usage on the new version is unavailable, it is difficult to use this information to facilitate call-graph analysis. It is an interesting work to factor in reflection to achieve safe regression test selection.

2) Non-Deterministic Executions

Android apps may use both threads and asynchronous tasks extensively for parallel processing. As a result, the execution orders of instructions in an app over a test case are non-deterministic due to scheduling non-determinism. Furthermore, the execution of Android application is also affected by environmental factors such as system-level events, network state changes, and volatile sensor data. A complete record of all such non-deterministic choices for deterministic replay is beyond the scope of this work. Another workaround to handle non-deterministic execution is to combine ReTestDroid with a deterministic replay tool, which is an active topic in concurrent testing research.

IV. EXPERIMENTAL STUDY

In this selection, we present an experimental study on five real-life open-source Android applications to evaluate ReTestDroid. We select both medium-scale and large-scale Android apps as subjects to conduct our experiment. In this way, we want to evaluate whether ReTestDroid is effective and practical when applied on real-life Android applications.

A. Subjects

We selected five real-life open-source Android apps for experimentation. *K9-Mail* is a popular email application. Music is the built-in music application of the official Android system. *Open Sudoku* is a game application. *Tomdroid* is a note application. *Mozilla Firefox Mobile* is a popular mobile Web browser. The descriptive statistics of the five applications are shown in Table 3. The rows list the five subject programs. The columns represent the name, description, number of program versions, average line of code of each subject, and the features used in the application related to our study. For example, *K9-Mail* is an email application. It has 6 program versions used in the experiment. It contains 24.5K lines of code. Finally, it uses both the *AsyncTask* and *Fragment* features in its implementation. The statistics of other subjects can be interpreted similarly. For each subject, we treat the first version as v_0 and number the subsequent versions as v_1, v_2, v_3, v_4, v_5, v_6, and v_7.

TABLE 3. SUBJECT PROGRAMS

Subject	Description	Versions	LoC	Features
K9-Mail	Mail	6	24.5K	AsyncTask Fragment
Music	Music	6	11.9K	AysncTask
Open Sudoku	Game	6	3.4K	AysncTask
Tomdroid	Note	6	4.9K	None
Mozilla Firefox Mobile	Browser	8	86.9K	AysncTask, Fragment, Native

B. Research Questions

We aim to answer the following two research questions in this experimental study.

Research Question 1 (RQ1): Can the improved ICFG of ReTestDroid enable safer regression test selection?

Research Question 2 (RQ2): Is the ReTestDroid tool effective in reducing the regression test suite size for Android application?

C. Experiment Setup

In this section, we present the setup details of our experimental study.

1) Test suites and Program Versions

To construct the test suite for each program, we used the Monkey tool shipped with Android 4.4.4 for test case generation. For each test suite, we generated 10 groups of test cases, and each group contains 20 test cases. The test cases of the same group had the same number of events and the number of events in each group was defined as 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, and 5000. To avoid generating repetitive test cases, we used different random seeds for Monkey for each of the 200 test cases. For each application, the corresponding test suite contains 200 test cases. Moreover, on average, it takes around 400 minutes to execute the test suite on a program version.

As stated above, all the subjects we chose were open source applications. In our experiment, we first download natural versions from their respective software repositories. *K9-mail*

had 6 versions ranges from *3.910* to *4.000*. *Music* has 6 versions. The first version was cm-10.1, and the follow-up versions were subsequent commit versions. All 6 versions of *Open Sudoku* were commit versions starting from version *1558f8bda545d408ab3b0700aa298b8aa5205ec2*. *Tomdroid* has 6 versions ranging from *0.6.0* to *0.7.5*. Finally, *Firefox Mozilla Mobile* has 8 versions starting from *FIREFOX_AURORA_47_BASE*.

However, the problem of using those natural versions in our experiment is that the randomly generated test cases can hardly trigger any failures on them. This is partially attributed to the fact that these versions are relatively stable versions and partially due to fault detection ability of the random test suite. Since we want to evaluate the safety of the ReTestDroid tool in selecting test cases, we need failure-triggering test cases. So we further manually inject mutants (i.e., mutation faults) into those natural program versions to create faulty versions. The mutation bugs injected including null pointer bugs, ANR bug with a busy UI thread, and intent bugs triggering ActivityNotFound exception. Finally, with a postmortem analysis, we measured the percentage of failed test cases for each program version of each subject, and the failure rate ranges from 3% to 12%.

2) Collecting code coverage information

The test case selection algorithm requires the code coverage matrix of the test cases on previous program version. In the experiment, we use the *EMMA* [48] tool to help collect code coverage information of the test cases for Java code. For each project, we add and realize an *Instrumentation* test class to help start the *Monkey* tool with predetermined number of events and seed to test the application, collect coverage information generated by *EMMA* periodically, and write the coverage information into the SD Card. For native code, we use the *gcov* tool extension for Android [51] to generate code coverage information. After executing the instrumented program, we use *lcov* to collect code coverage information. Finally, we parse the code coverage file generated by *EMMA* and *lcov* to generate the coverage matrix of each test case for each program version for test case selection.

3) Experiment Procedure

We use a device model of *Meizu MX3* running on *Android 4.4.4* to perform the experiment. The testing host is a workstation with i7 quad-core processor and 16GB of memories.

Each subject had 6 or 8 natural program versions. Thus, in the experiment, we had 5 or 7 pairs of consecutive program versions (as $<P, P'>$): $<v_0, v_1>$, $<v_1, v_2>$, $<v_2, v_3>$, $<v_3, v_4>$, $<v_4, v_5>$, etc. For each subject program, and for each program version P, we executed each test suite against P to record its execution results and its coverage matrix. For RQ1, we first construct the ICFGs for each pair of P and P' with both FlowDroid and ReTestDroid. Then we perform impact analysis and regression test selection with the same algorithms realized in ReTestDroid on the two ICFGs, respectively. Then we measure the number of selections that are safe for each subject, (i.e., whether all failure revealing test cases are all selected). For RQ2, we measured the percentage of a test suite that was selected by ReTestDroid over each pair of program versions.

D. Results and Analysis

In this section, we present the results of our experimental study as well as the detailed analysis.

1) Answering RQ1

In this section, we want to answer whether the improved ICFG of ReTestDroid enable safer regression test selection than directly using the ICFG of FlowDroid.

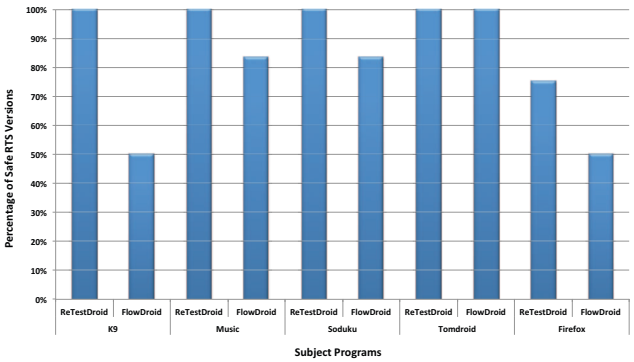


Fig. 4. Comparison of the Percentage of Safe RTS Versions

As shown in Fig. 4, for each subject program, we measured the percentage of program versions where RTS is safe based on the ICFGs of ReTestDroid, and the percentage of program versions where RTS is safe based on the ICFGs of FlowDroid. We can see that in general, the ICFGs of ReTestDroid can enable safer regression test selection than that of FlowDroid. For 4 out of the 5 subject programs, ReTestDroid is safe on 100% of the program versions. In contract, RTS on the ICFG of FlowDroid is only 100% safe on all versions of Tomdroid. In fact, only 50%, 83%, 83% of the versions are safe when performing RTS on ICFG of FlowDroid for K9-Mail, Music, and Open Sudoku. We analyzed the subject program versions and the test suites selected, and we found the asynchronous task and fragment features lead to the unsafe RTS selection on FlowDroid. On Tomdroid, the RTS on the ICFG of both ReTestDroid and FlowDroid are safe, which is as expected since Tomdroid is a relatively simple application without much features.

On the other hand, Firefox mobile is the only application on which the RTS on both ICFGS of ReTestDroid and FlowDroid are unsafe. If we examine the versions carefully, RTS is safe on 6 out of the 8 programs versions based on the ICFGs of ReTestDroid. We checked the test cases carefully, and found the non-deterministic execution lead to the unsafe selection on these two versions by ReTestDroid. In contrast, the RTS based on the ICFGs of FlowDroid is only safe on 50% of the versions of Firefox mobile. After a close examination of the artifacts, we found the RTS on the ICFGs of FlowDroid is not only affected by non-deterministic executions, but is also affected by AsynTask, Fragment, and native code features.

Therefore, we can answer RQ1 that the ICFGs of ReTestDroid can enable safer regression test selection than that of the FlowDroid.

2) Answering RQ2

In this section, we want to answer whether our ReTestDroid system is effective in reducing the regression test suite size for Android application.

The percentages of test cases selected for each pair of program versions are shown in Fig. 5. In the plot, each bar is associated with a version. For example, v_1 of *K9* means that percentage of test cases selected for the pair $<v_0, v_1>$ of *K9*-mail. Other bars can be interpreted similarly.

We can see that the percentage of the test cases selected ranges from 12% to 72% across the subject programs. The average percentages of test cases selected are 44.8%, 39.4%, 44.3%, 40.9%, and 27.4% for the 5 subjects, respectively. It shows that our *ICFGs* are sensitive to changes so that many test cases that are modification revealing can be identified. The saving in terms of the percentage of test cases not selected is high. Therefore, we can answer *RQ2* that the ReTestDroid system is effective in reducing the size of a given test suite for regression testing of Android applications.

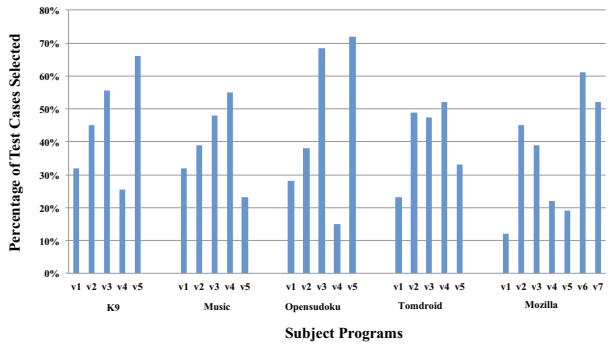


Fig. 5 Percentage of Test Cases Selected for Each Version

We further measured the overall regression testing time for ReTestDroid. We found the time for impact analysis was much shorter than the time for test case execution. As a result, there is significant timesaving in terms of total regression testing time with regression test case selection.

3) Threats to Validity

In our experimental study, we use only Monkey tool to generate random test suites for regression testing. A further study on regression test suites generated by other test case generation tools or designed by human testers will further strengthen the validity of our study. Another factor affecting the threat to validity is the correctness of our platform. We used Java and C to implement our ReTestDroid system for experimental study. We have carefully performed code review and testing on our platform to ensure their correctness.

We used 5 open source applications to evaluate our ReTestDroid platform. While they cover some types of Android application, a thorough study on more types of popular Android applications will also improve the validity of our study.

V. Related Work

In this section, we systematically present the closely related works.

The Regression Test Selection (RTS) techniques are originally studied for procedural programs [21][23] and then are targeted at Object-Oriented programs [6][27] and domain specific programs [39]. When classified based on the program analysis technique used, RTS techniques can fall into the dataflow analysis based techniques, the control flow based techniques, firewall-based techniques, and differencing-based techniques, etc.

Dataflow analysis-based RTS techniques detect definition-use pairs for variables to describe program modifications and select test cases that cover the paths impacted by modified variables. Harrold and Soffa [30][31] extended data flow testing to guide the selection and execution of test cases, their study can be applied to analyze changes across multiple procedures.

Control flow analysis-based techniques [21][4] analyze control flow models of the input programs. Rothermel and Harrold proposed a safe, efficient regression test selection technique [35]. Their algorithms construct control flow graphs for a program and its modified version and use these graphs to select tests that execute impacted code from the original test suite, and these algorithms are safe under certain constraints. Rothermel et al. [36] proposed a control flow analysis-based technique for C++ programs. The Inter-procedural Control Flow Graph (ICFG) and Class Control Flow Graph (CCFG) are proposed to model programs. These algorithms select relevant regression test cases by comparing graph models of the original and the modified program. And these authors have also presented several other works on control flow analysis-based RTS techniques [32][33][34]. Ball et al. [4] focus on the application of control flow analysis and control flow coverage to the regression test selection problem, considering how the type of coverage information collected can affect the precision of regression test selection algorithms. They reformulated Rothermel and Harrold's regression test selection algorithm and presented three new algorithms.

Leung and White [21] proposed a firewall-based RTS technique, and the firewall-based approach presents regression testing of modules where dependencies due to both control flow and data flow are taken into consideration. Kung et al. [18][19][20] proposed the firewall-based RTS technique for object-oriented programs. They used three models to represent the dependencies of a C++ program: Object Relation Diagram (ORD), Block Branch Diagram (BBD), and Object State Diagram (OSD). Jang et al. [16] also proposed a RTS algorithm for C++ programs. Their change impact analysis approach constructs a method-level firewall and aims at identifying all affected methods efficiently.

TestTube is a system that combines static and dynamic analyses to perform selective retesting of software systems written in C. It uses the Differencing-Based Technique to select test cases [9]. Vokolos and Frankl also proposed a differencing-based technique that was based on a textual differencing of two programs. This technique converts a

program to a canonical form, which can avoid trivial differences between programs and compares canonical versions to detect modifications.

There are also RTS techniques for domain specific application. Regression test selection techniques for database applications face challenges that database applications are not stateless and test cases may affect each other. Willmor and Embury [39] proposed a safe selection algorithm for database applications by extending the control flow analysis-based algorithm of Rothermel and Harrold [35]. They introduced the concept of database dependencies to select database-dependent test cases with respect to the database state.

Different from the above works, ReTestDroid addresses the safe regression test selection challenges raised by the Android programming model.

In [13], Do et al. proposed a similar regression test selection approach for Android application. Our work differs from them in two aspects. First, our approach proposes a more precise ICFG for android application to accommodate Android features such as fragment, native code, and asynchronous tasks. Second, we perform a comprehensive experimental study to compare the safety of ReTestDroid with existing approach. The results show ReTestDroid is safer for effective regression test selection.

There are many works on testing techniques for Android apps. Monkey [53] is the most frequently used tool for performing random testing on Android application. It randomly generates UI events and considers the app as black-box. Dynodroid [27] also uses a random event generation strategy but is more efficient when compared to Monkey. It can generate system events and implements the traversal strategy in a smarter way. Other Android test case generation tools like GUIRipper [1] and SwiftHand [10] et al. use Model-based exploration strategy to generate events and explore the behavior of the application systematically. They usually build models dynamically and iteratively to explore states triggered from a discovered one. In this work, we focus on the safe regression test selection problem for Android apps rather than the test case generation problem.

VI. Conclusion and Future Work

When a mobile application is changed, testers can conduct regression testing to ensure the changes made to the application have no adverse effect. In this work, we have proposed a regression test selection system ReTestDroid for Android application. Building on top of the interprocedural control flow graph generated by FlowDroid, ReTestDroid constructs a novel interprocedural control flow graph for Android apps to accommodate important distinct features of Android programming model, including Fragment lifecycle, native code, asynchonrous background tasks. Our experimental results on 5 real-life Android applications have shown that the improved ICFG proposed by ReTestDroid are effective to support safer regression test selection. Furthermore, our ReTestDroid tool can also significantly reduce test suite size and regression testing time, which can be practical for use.

References

[1] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., Memon, A. M. Using GUI Ripping for Automated Testing of Android Applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 258–261, New York, NY, USA, 2012.

[2] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y., Octeau, D., McDaniel, P.FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14). ACM, New York, NY, USA, 259-269, 2014.

[3] Bacon, D. F., Sweeney, P. F. Fast static analysisof C++ virtual function calls. In Proceedings of the11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 324-341, Oct. 1996.

[4] Ball, T. On the limit of control flow analysis for regression test selection. In ACM Internationl Symposium on Software Testing and Analysis, pages 134-142, 1998.

[5] Bible, J., Rothermel, G., Rosenblum, D. A comparative study of coarse- and fine-grained safe regression test selection. ACM Transactions on Software Engineering and Methodology, 10(2):149-183, 2001.

[6] Binder, R. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, 1999.

[7] Biswas, S., Mall, R., Satpathy, M., Sukumaran, S. Regression Test Selection Techniques: A Survey. Informatica (Slovenia) 35(3): 289-321, 2011.

[8] Cheatham, T., Mellinger, L. Testing object-orientedsoftware systems. In Proceedings of the Computer Science Conference, pages 161-165, 1990.

[9] Chen, Y. F., Rosenblum, D. S., Vo, K. P. TestTube: A system for selective regression testing. In Proceedings of the 16th International Conference on Software Engineering, pages 211-222, 1994.

[10] Choi, W., Necula, G., Sen, K. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13), pages 623–640, New York, NY, USA, 2013.

[11] Cooper, B. F., Lee, H. B., Zorn, B. G. Profbuilder: A package for rapidly building JAVA execution profilers. Technical report, University of Colorado.

[12] Dean, J., Grove, D., Chambers, C. Optimizationsof object-oriented programs using static class hierarchyanalysis. In European Conference on Object-Oriented Programming, pages 77-101, 1995.

[13] Quan Chau Dong Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. Redroid: A Regression Test Selection Approach for Android Applications. In Proceedings of The 28th International Conference on Software Engineering and Knowledge Engineering, Redwood City, San Francisco Bay, USA, pages 486-491, 2016.

[14] Graves, T., Harrold, M. J., Kim, J.-M., Porter, A., Rothermel, G. An empirical study of regression test selection techniques. In Proceedings of the International Conference on Software Engineering, pages 188-197, 1998.

[15] Hsia, P., Li, X., Kung, D., Hsu, C-T., Li, L., Toyoshima, Y., Chen, C.: A technique for the selective revalidation of OO software. Software Maintenance: Research and Practice, 9:217-233, 1997.

[16] Jang, Y., Munro, M., Kwon, Y. An improved method of selecting regression tests for C++ programs. Journal of Software Maintenance: Research and Practice, 13(5):331–350, 2001.

[17] Kim, J.-M., Porter, A., Rothermel, G. An empiricalstudy of regression test application frequency. In Proceedings of the 22nd International Conference on Software Engineering, pages 126-135, 2000.

[18] Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., Kim, Y-S., Song, Y-K. Developing an object-orientedsoftware testing and maintenance environment. Communications of the ACM, 38(10):75-87, 1995.

[19] Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., Chen, C. On regression testing of object-oriented programs. The Journal of Systems and Software, 32(1):21-40, 1996.

[20] Kung, D., Gao, J., Hsia, P., Wen, Y., Toyoshima, Y. Change impact identification in object-oriented software maintenance. In Proceedings of the International Conference on Software Maintenance, pages 202-211, 1994.

[21] Laski, J., Szermer, W. Identification of program modifications and its applications in software maintenance. In Proceedings of the Conference on Software Maintenance, pages 282–290, 1992.
[22] Leung, H. K. N., White, L. J. A study of integrationtesting and software regression at the integration level. In Proceedings of the Conference on Software Maintenance, pages 290-300, 1990.
[23] Leung, H. K. N., White, L. J. Insights into testingand regression testing global variables. Journal of Software Maintenance: Research and Practice, 2:209-222,1990.
[24] Leung, H. K. N., White, L. J. A cost model to compare regression test strategies, In Proceedings of International Conference on Software Maintenance, Sorrento, 1991, pp. 201-208.
[25] Lam, P., Bodden, E., Hendren, L. The Soot framework for Java program analysis: a retrospective. In Cetus Users and Compiler Infrastructure Workshop (CETUS 2011), 2011.
[26] Liang, D., Pennings, M., Harrold, M. J. Extendingand evaluating flow-insensitive and context-insensitivepoints-to analyses for JAVA. In Proceedings of the ACM Workshop on Program Analyses for Software Tools andEngineering, 2001.
[27] Machiry, A., Tahiliani, R., Naik, M. Dynodroid: An Input Generation System for Android Apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013.
[28] McGregor, J., Sykes, D. A Practical Guide to Testing Object-Oriented Software. Addison-Wesley, 2001.
[29] Harrold, M. J., Jones J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, A., Gujarathi, A. Regression test selection for Java software. In Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01). ACM, New York, NY, USA, 312-326, 2001.
[30] Harrold, M., Soffa, M. Interprocedural data flow testing. In Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification, pages 158–167, 1989.
[31] Harrold, M., Soffa, M. An incremental approach to unit testing during maintenance. In Proceedings of the International Conference on Software Maintenance, pages 362–367, October 1988.
[32] Rothermel, G., Harrold, M. J. Selecting regression tests for object-oriented software. In International Conference on Software Maintenance, pages 358-367, 1993.
[33] Rothermel, G., Harrold, M. J. Analyzing regression test selection techniques. IEEE transactions on Software Engineering, 22(8):529-551, 1996.
[34] Rothermel, G., Harrold, M. J. A safe, efficient regression test selection technique. ACM Transactions onSoftware Engineering and Methodology, 6(2):173-210, 1997.
[35] Rothermel, G., Harrold, M. J. Empirical studies of a safe regression test selection technique. IEEE Transactions on Software Engineering, 24(6):401-419, Jun.1998.
[36] Rothermel, G., Harrold, M. J., Dedhia, J. Regression test selection for C++ software. Journal of Software Testing,Verication, and Reliability, 10(6):77-109, 2000.
[37] Rothermel, G., Harrold, M. J. Analyzing regression test selection techniques. IEEE Transactionson Software Engineering, 22(8):529-551, 1996.
[38] Tip, F., Palsberg, J. Scalable propagation-based callgraph construction algorithms. In Proceedings of theConference on Object-Oriented Programming Systems, Languages, and Applications, 281-293, Oct. 2000.
[39] Willmor, D., Embury, S. A safe regression test selection technique for database-driven applications. In Proceedings of the 21st IEEE International Conference on Software Maintenance, pages 421–430. IEEE Computer Society, 2005.
[40] Yoo, S., Harman, M. Regression testing minimization, selection and prioritization: a survey. SoftwareTesting, Verification and Reliability, 1(1):121–141, 2010.
[41] Android application updates frequency. https://www.nowsecure.com/blog/2015/06/09/understanding-android-s-application-update-cycles/
[42] Harvard Univeristy. Interprocedural Analysis. http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec05-Interprocedural.pdf
[43] After update - chrome keeps crashing. https://productforums.google.com/forum/#!topic/chrome/bGsPLTKmBpM
[44] Android NDK. https://developer.android.com/ndk/index.html
[45] AsyncTask. https://developer.android.com/reference/android/os/AsyncTask.html
[46] Building a Dynamic UI with Fragments. https://developer.android.com/training/basics/fragments/index.html
[47] Contacts app crashing after OTA update 23.3.24. http://forums.androidcentral.com/motorola-droid-maxx/465113-contacts-app-crashing-after-ota-update-23-3-24-a.html
[48] Emma. http://emma.sourceforge.net/
[49] Fragments. https://developer.android.com/guide/components/fragments.html
[50] Graphviz. http://www.graphviz.org/Download.php
[51] Locv extension for Android. https://github.com/spbnick/lcov-android/
[52] LLVM's Analysis and Transform Passes. http://llvm.org/releases/2.5/docs/Passes.html
[53] The Monkey UI android testing tool. http://developer.android.com/tools/help/monkey.htm