Redroid: A Regression Test Selection Approach for Android Applications

Quan Do*, Guowei Yang*, Meiru Che†, Darren Hui†, Jefferson Ridgeway§
* Department of Computer Science, Texas State University, San Marcos, TX 78666, USA
† Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712, USA
§ Elizabeth City State University, Elizabeth City, NC 27909, USA
Email: * {q_d2, gyang}@txstate.edu, † {meiruche, darren_hui}@utexas.edu, § jdridgeway787@students.ecsu.edu

***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. However, researchers have primarily considered testing of single version of mobile applications. Although regression testing has been extensively studied for desktop applications, and many efficient and effective approaches have been proposed, these approaches cannot be directly applied to mobile applications. We first present a bug study on real-world Android bugs to show the existence of regression bugs, which motivates the need for an efficient regression test selection technique for Android applications. Next, we introduce Redroid, a new approach to regression test selection for Android applications. Our approach leverages the combination of static impact analysis and dynamic code coverage, and identifies a subset of test cases for re-execution on the modified application version. We implement our approach for Android applications, and demonstrate its efficacy through an extensive empirical study.**

I. 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 applications (or simply, "apps") in terms of portability, reliability and security. In recent years, a great deal of research has been performed to improve the reliability of mobile apps on mobile platform, for example, by applying automatic testing [5], [15], [6], [19], [8], [16], [7].

Mobile apps are evolving over time, for example, to cope with new requirements or to fix bugs, and regression testing—a process of validating modified software to ensure that changes are correct and do not adversely affect other features of the software—needs to be performed on the new versions of the mobile apps. However, the majority of the research is focused on testing of single version of mobile apps. Although regression testing has been extensively studied for desktop applications, and many efficient and effective approaches have been proposed [10], [18], [13], [17], [11], [12], [20], these approaches cannot be directly applied to mobile apps. A key factor that causes the incompatibility is the difference between the mobile platform's system architecture and the desktop platform's. For example, although Android apps are developed using the Java language, they use the Dalvik Virtual Machine as a runtime environment, which is significantly different from the Java Virtual Machine.

This paper first presents a bug study based on 10 real-world Android apps from Google Code Repository [3]. The study shows that there are regressions for Android apps during evolution, and an efficient and effective regression testing approach is highly needed for this area. This paper then introduces `Redroid`, a new approach to regression test selection for Android apps. Given a test suite that was performed on the original Android app, and the two versions involved in a change, `Redroid` identifies a subset of the tests that must be re-executed to test the new Android version. Leveraging the combination of static impact analysis with coverage information that is dynamically generated at runtime, our approach identifies a subset of tests to check the behaviors of the modified version that can potentially be different from the original version. We developed a prototype tool for `Redroid`, and conducted an evaluation based on two real-world Android apps, which shows that our approach can significantly reduce the number of tests for re-execution after an Android app is modified.

The remainder of this paper is organized as follows: we present a bug study on open source Android apps in Section II. We present our approach in Section III, and then evaluate it in Section IV. We discuss related work and conclude the paper in Sections V and VI.

II. Bug Study

We conducted a study to investigate real-world Android bugs with an aim to find how these change-related bugs (regressions) are manifested in Android apps.

We selected Android apps from Google Code Repository [3] based on four specific criteria: 1). large number of downloads, 2). large number of bug reports, 3). long development history, and 4). wide range of apps from different categories. The Google Code Repository houses over 900 apps, and enables users to give feedback on their apps and provide insight to bugs that the app may have while developers are unaware of. The feedback or bug reports that users submit to Google Code Repository for the specific app are given labels and types as that bug is being worked on by the developer(s). Table I lists the 10 apps that were selected for this study. The number

DOI reference number: 10.18293/SEKE2016-223

TABLE I
BUG STUDY RESULTS BASED ON REAL-WORLD ANDROID APPS FROM GOOGLE CODE REPOSITORY.

Application	Category	# Downloads	# Bug Reports	Time Span (yrs)	# Regressions
Ankidroid	Education	1,000,000-5,000,000	2,645	6	15
ConnectBot	Communication	1,000,000-5,000,000	688	6	2
Android Wifi Tether	Communication	380,000	1,965	4	5
Ebookdroid	Productivity	1,000,000-5,000,000	937	4	10
Android SMS	Tool	70,000	195	5	4
Android Shuffle	Productivty	50,000-100,000	330	5	6
Android Privacy Guard	Communication	100,000-500,000	166	2	3
Open GPS Tracker	Travel	100,000-500,000	432	4	1
Electric Sleep	Health	100,000-500,000	217	2	3
DroidWall	Tool	1,000,000-5,000,000	318	3	10

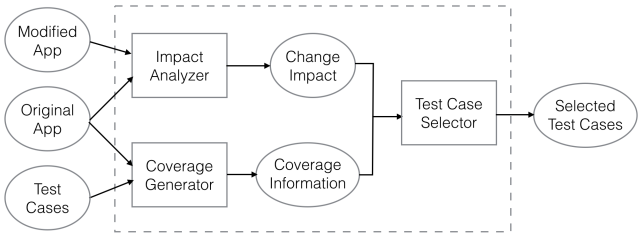

Fig. 1. Redroid Overview.

of downloads and category per app are retrieved from the Google Play store. These 10 apps, chosen from six categories, have 2-6 years of history, and all of them have more than 50k downloads. All of these apps have more than a hundred bug reports, and some of them have over two thousand bug reports.

For each reported bug, there are eight possible labels : Invalid, WontFix, Duplicate, Accepted, Started, Fixed, Fix-Pending, and Verified. The types of each bug report include enhancements and defects. We manually analyzed bug reports with the label "Fixed" and the type "Defect", which also signifies that the issue has been solved with an updated version of the app and that the app did have a bug. We found that regression bugs do exist in these apps. In particular, 15 regression bugs are found in Ankidroid. Note that our analysis may not find all regression bugs in these apps as we require the bug reports to explicitly mention that the bug is caused by changes, such as changes to the app's source code, to library (i.e., a version update of the Android operating system), to app's configuration settings, or to hardware configurations.

We also analyzed these bugs with an aim for bug replication; however, it turns out that replication is difficult since the quality of the report is poor even though there were 10 strong app candidates, due to one or more of the following conditions: insufficient information of replication of the bug and/or how the bug was fixed, the developer(s) not providing the apk file and/or source code files on the Google Code Repository or Github repository, or the source code was not able to be compiled in either the Eclipse Integrated Development Environment (IDE) or Android Studio.

This study shows that there are regressions for Android apps during evolution, and an efficient and effective regression testing approach is highly needed for this area.

III. APPROACH

A. Overview

Given two Android app versions and an initial test suite, `Redroid` automatically computes the test cases that need to be re-executed on the modified app version. An overview of the approach is shown in Figure 1. There are three main components in the framework: `impact analyzer`, `coverage generator` and `test case selector`.

The `impact analyzer` takes as input the original app and its the modified version, and generates impacted code that contains changes between the two versions. It uses the algorithm in `Dejavu` [18] for computing the change impact. `Dejavu` constructs control-flow graph (CFG) representations of the methods in the two app versions P and P', in which individual nodes are labeled by their corresponding statements. Following identically-labeled edges, Dejavu performs a simultaneous depth-first graph walk on a pair of CFGs G and G' for each method and its modified version in P and P' to find code changes. Given two edges e and e' in G and G', if the code associated with nodes reached by e and e' differs, e is called a dangerous edge as it leads to code that may cause program executions to exhibit different behavior. However, a special handling is needed to find pairs of anonymous inner classes (AICs) in two app versions. Different from regular Java procedures, which are defined as a separated method, AIC is defined as an inner procedure of another procedure.

The `coverage generator` collects coverage information while the original test suite is executed on the original app version. It computes the lines of code that are executed by each test case. While there are tools such as `EMMA` [2] that can be used to compute the code coverage during Android testing, we find that `EMMA` cannot provide the coverage information for each individual test case when a set of test cases are executed in one time. In other words, to get the coverage for each test case, we have to build and run one test case at a time. Therefore, in this work, we automatically instrument each block in bytecode level, so that when executed the intrumented code generates some execution logs, where we can identify the blocks executed by each test.

`Test case selector` takes as input the change impact information from `impact analyzer` and the coverage information from `coverage generator`, and selects for re-

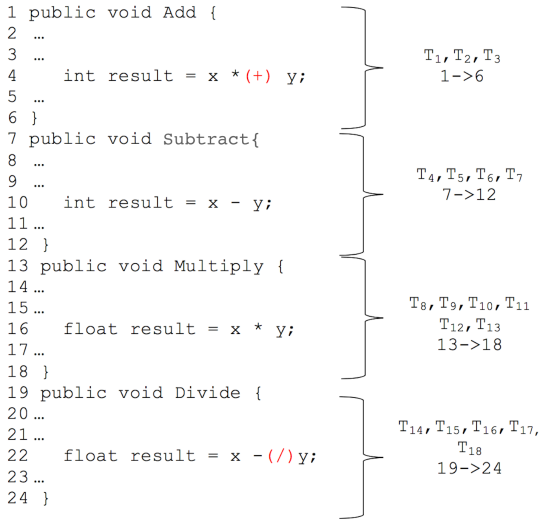

Fig. 2. The four arithmetic operations and their corresponding tests.

execution test cases that are affected by the changes. Since the `impact analyzer` outputs changed blocks using the same format as used by the `coverage generator`, the `test case selector` can easily check whether a test case executes any changed block; if so, that test case is selected for re-execution, as this test case can execute the change and thus its execution on the modified app version may exhibit different behaviors.

B. Illustrating Example

Simple Calculator is a basic calculator that takes as input two numbers and performs four arithmetic operations: addition, subtraction, multiplication and division, which correspond to buttons "+", "−", "∗", and "/" respectively. Figure 2 shows part of its implementation for these four operations. 18 test cases are created to test the app's functionality. By running all the 18 test cases, two bugs are found in the original app, and two fixes are made to lines 4 and 22, where the operators "∗" and "−" are replaced by "+" and "/" (in the parentheses), respectively.

After modifying an implementation, the app is required to be re-tested in order to assure the two fixes remove the bugs that are found previously and do not adversely affect other program behaviors. Traditional approach simply re-executes all the test cases in the original test suite on the modified app. However, the modifications may not impact all the test cases, and simply re-running all test cases may not be an efficient way to check the modified app.

Our approach combines change impact analysis and coverage information to reduce the number of test cases for re-execution, and only re-executes the test cases that can potentially reveal different behaviors from the original app version. First, by running all the test cases in the original test suite on the original app, our approach uses coverage generator to collect the coverage information of each test case, i.e., the lines of code that are executed by each test case. As shown in Figure 2, lines $1 \rightarrow 6$ are covered by test cases T_1, T_2, and T_3; lines $7 \rightarrow 12$ are covered by test cases T_4, T_5, T_6, and T_7; lines $13 \rightarrow 18$ are covered by test cases T_8, T_9, T_{10}, T_{11}, T_{12}, and T_{13}; and lines $19 \rightarrow 24$ are covered by test cases T_{14}, T_{15}, T_{16}, T_{17}, and T_{18}. Moreover, impact analyzer is used to find code changes, and it finds that lines 4 and 22 are changed. Combining this change information and the computed coverage information, our approach selects test cases T_1, T_2, T_3, T_{14}, T_{15}, T_{16}, T_{17}, and T_{18} for re-execution on the modified app, since the execution traces of these test cases contain the changes. It eliminates 10 out of 18 test cases for re-execution, which is a 55.6% reduction.

TABLE II
ANDROID APPS SELECTED FOR EVALUATION. EACH APP HAS TEST CASES MAINTAINED BY THE DEVELOPERS FOR TESTING.

Apps	Classes	Methods	LOC	Versions	Test Cases
Inetify	63	356	1,500	15	206
AndStatus	250	2,700	15,000	15	99

IV. EVALUATION

Since our technique relies on the output of the `coverage generator (CG)`, we first evaluate the efficiency and preciseness of the `CG`, by considering two research questions:

- **RQ1:** How efficient is `CG` compared to the exiting coverage analysis tool `EMMA`?
- **RQ2:** Does `CG` achieve the same level of preciseness compared to `EMMA`?

We then evaluate the overall cost of `Redroid` relative to the traditional approach which re-runs all test cases (`ReTestAll`) by considering two research questions:

- **RQ3:** How does the number of test cases selected by `Redroid` compare to the traditional `ReTestAll` approach?
- **RQ4:** How does the time cost of applying `Redroid` compare to the traditional `ReTestAll` approach?

A. Artifacts

Although 10 apps were studied for our bug study in Section II, no tests were found for these apps. Thus, we selected two other open source Android apps for evaluation: `Inetify` [4] and `AndStatus` [1]. For each of the two selected apps, Table II provides information on its associated number of class files (`Classes`), number of methods (`Methods`), number of lines of code (`LOC`), number of basic blocks (`Blocks`), number of versions we used for evaluation (`Versions`), and number of test cases. The test cases are found from the projects and maintained by the developers for testing the functionalities of these two apps.

`Inetify` 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 Wifi otherwise. `AndStatus` allows users to login multiple social app accounts such as Twitter and Pump.io. It can combine multiple accounts from all networks into one Timeline and allow users to read and post even if they are offline.

To evaluate our approach, we required multiple versions of each app being analyzed. Because multiple versions of these apps were not available, we generated versions by manually creating mutants of the available app (v0). When creating mutants, we considered a broad range of changes that can be applied to the code: *change type*, *change location*, and *number of changes*. The change types include modification, deletion, and addition of source code statements. Changes are introduced at different locations: (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). AIC procedures are specially considered because they are special Android features which are not existent in regular Java programs, and they are generated as separate class files by the Android Development Environment. Each mutant has one, three, and five changes. For each app, we created five mutants v_1 to v_5 each with one change, five mutants v_6 to v_{10} each with three changes, and five mutants v_{11} to v_{15} each with five changes. Thus, in total we created 15 versions for `Inetify` and 15 versions for `AndStatus`, respectively.

B. Variables and Measures

1) Independent Variables: The independent variables that we used in the empirical study are the different techniques used for computing coverage and for selecting test cases. To study RQ1 and RQ2, we use our `coverage generator (CG)`, which is based on instrumentation, and the existing coverage tool `EMMA` [2], a built-in testing framework for Android Platform. To study RQ3 and RQ4, we use `Redroid` and the traditional `ReTestAll` approach which re-tests all test cases.

2) Dependent Variables: We selected three dependent variables and measures which are ultimately related to the preciseness and cost of coverage generation techniques and test case selection techniques. Given an original app *P*, which is modified to a new version *P'*. Note that the costs can be measured differently depending on what technique is being applied. `CG` and `EMMA` are only applied to the original app version *P*. `Redroid` is applied to both `P` and `P'`.

The first dependent variable is *execution time*. To study RQ1, we need to measure execution time to compare the efficiency of `CG` versus `EMMA`. To study RQ4, we also need to measure execution time to compare the time cost of `Redroid` and `ReTestAll`. The time cost of `Redroid` is divided to the time for performing static impact analysis, coverage generation, and test selection—which is considered as the overhead of applying `Redroid` and the time for executing the selected test cases.

The second dependent variable is *code coverage*, which is used to study RQ2. It is measured in terms of the number of executed blocks and the percentage of executed blocks out of the total number of blocks.

The third dependent variable is *the number of selected test cases*. To study RQ3, we need to measure the number of selected test cases for each test case selection technique, since the goal of our approach is to reduce the number of test cases and thus reduce the cost of performing regression testing. We note that reduction in number of test cases does not necessarily correlate with reduction in time cost, because not all test cases are the same in terms of their execution cost. For example, some test case's execution may cover a large portion of an application and may take long time to execute; while some test case's execution may cover a small portion of the application and may take short time to execute.

C. Experiment Setup

We ran `Redroid` using the Android Development Environment, together with the Ant tool and Eclipse. The study was performed on a Windows operation system running at 3.4GHz with 8GB of memory. We automated the process from analyzing change impact, generating code coverage report, to selecting test cases by running a batch file under Windows operation system.

For each original app version v_0, we ran all the test cases in the original test suite with the application of `CG` and `EMMA`.

For each mutant app version v_k ($k > 0$), we performed `Redroid`, which selects test cases by analyzing v_k and v_0, and runs only the selected test cases on v_k; we also performed `ReTestAll`, which re-runs all the test cases in the original test suite on v_k.

D. Results and Analysis

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

Table III presents the results of applying the two code coverage analysis tools `CG` and `EMMA`. In the table, for each app we list the total number of blocks, the number of executed blocks, the percentage of executed blocks out of the total blocks, and the time cost for coverage analysis, computed by `CG` and `EMMA`, respectively.

Table IV presents the results of applying the two test case selection techniques `ReTestAll` and `Redroid` on `Inetify`. In the table, for each mutant version, we list the number of selected test cases and the time cost for running these tests using `ReTestAll`; we also list the overhead caused, the number of selected test cases and the time cost for executing these test cases using `Redroid`.

Table V presents the results of applying the two test case selection techniques `ReTestAll` and `Redroid` on `AndStatus`. We list similar types of results as in Table IV.

RQ1: How efficient is `CG` compared to the exiting coverage analysis tool `EMMA`?

In Table III, we can see that, for `Inetify`, `EMMA` took 341 seconds to get the coverage information for each test case in the test suite, while `CG` only took 136 seconds, which is more than 60% reduction in time cost. While for `AndStatus`, the reduction is not that much, `CG` is still more efficient than `EMMA`. The main reason for the reduction achieved by `CG` is because

TABLE III
RESULTS OF CG VS. EMMA

Apps	Number of Test Cases	CG				EMMA			
		Total Blocks	Excuted Blocks		Time (SS)	Total Blocks	Excuted Blocks		Time (SS)
Inetify	206	7,403	4,930	66.6%	136	6,691	4,452	66.5%	341
AndStatus	99	71,304	41,071	57.6%	497	65,904	37,770	57.3%	595

TABLE IV
TEST CASE SELECTION RESULTS FOR INETIFY

Versions	# Changes	Redroid		
		Overhead (SS)	# Test Cases	Time (SS)
v1	1	4	56	37
v2	1	4	38	25
v3	1	4	39	26
v4	1	4	36	24
v5	1	4	53	35
v6	2	4	108	71
v7	2	4	63	41
v8	2	4	95	62
v9	2	4	111	73
v10	2	4	86	56
v11	3	4	136	96
v12	3	4	140	91
v13	3	4	126	82
v14	3	4	162	106
v15	3	4	171	112

TABLE V
TEST CASE SELECTION RESULTS FOR ANDSTATUS

Versions	# Changes	Redroid		
		Overhead (SS)	# Test Cases	Time (SS)
v1	1	6	8	40
v2	1	6	9	45
v3	1	6	4	20
v4	1	6	10	50
v5	1	6	8	40
v6	2	6	40	200
v7	2	6	37	185
v8	2	6	18	90
v9	2	6	46	230
v10	2	6	44	220
v11	3	6	55	275
v12	3	6	67	335
v13	3	6	64	320
v14	3	6	70	350
v15	3	6	66	330

using EMMA we have to build and run each test at a time to compute its code coverage.

RQ2: Does CG achieve the same level of preciseness compared to EMMA?

In Table III, we can see that CG generated $7,403$ blocks while EMMA generated $6,691$ blocks, which is a 712-block difference for Inetify. Moreover, CG generated $71,304$ blocks and EMMA generated $65,904$ blocks, which is a $5,400$-block difference for AndStatus. One possible reason for this difference is that during the process of instrumentation, we insert some virtual blocks such as entry blocks and ending blocks for each CFG, which can lead to more blocks to be counted. Since we do not have knowledge of EMMA's implementation, we are unable to verify what kind of measurement is used by EMMA to count the number of total generated blocks.

Similarly, we can also find the difference in the executed blocks for both Inetify and AndStatus.

Despite the difference in the number of total blocks and in the number of executed blocks, CG and EMMA achieved almost the same code coverage in terms of percentage. We find a 0.1% difference in code coverage for Inetify, and a 0.3% difference in code coverage for AndStatus. These differences are so small that they can be neglected. Even if they cannot be neglected, since CG provides more coverage than EMMA, CG can be considered as conservative if not as precise as EMMA, therefore, it is safe to use CG's output for computing test cases.

RQ3: How does the number of test cases selected by Redroid compare to the traditional ReTestAll approach?

From Table IV, we can see that for Inetify, while ReTestAll selected all the 206 test cases, Redroid selected a range from 36 to 172 test cases, achieving 16% to 83% reduction in the number of selected test cases. Moreover, we find the the number of selected test cases varies for mutant versions with the same number of changes, e.g., for mutant versions v_0 to v_5 where only one change is involved in each mutant, Redroid selected five different numbers of test cases. This shows that different changes can actually have different impact on behaviors of the app, and thus lead to different selections of test cases. Despite that, unsurprisingly we find that overall the more changes got involved in the mutants, the more test cases were selected, since usually more changes would have more impact. Similar pattern of results can be also found in Table V.

RQ4: How does the time cost of applying Redroid compare to the traditional ReTestAll approach?

From Table IV, we can find that for Inetify, there was 4 seconds overhead of applying Redroid, due to static impact analysis, coverage generation, and test selection. However, compared the execution time of all the test cases which is 134 seconds, the overhead is not much. Similarly, from Table V we find that for AndStatus there was 6 seconds overhead compared to 495 seconds for executing all the test cases.

Moreover, we see that when Redroid is performed, due to the reduction of the number of test cases for re-execution, the time cost for executing the selected test is also reduced. Even when the overhead is counted, Redroid still achieved reduction in time cost. The biggest time reduction is 79% for version v_4 in Inetify, and 95% for version v_3 in AndStatus, respectively.

V. Related Work

Android bugs have been studied previously for different purposes. For example, Hu and Neamtiu [14] conducted a bug study to investigate the categories of Android bugs and how Android bugs are manifested; Bhattacharya et al. [9] performed a bug study to understand the bug-fixing process in Android platform and Android-based apps; and Zaeem et al. [21] performed a study to identify user-interaction features for which oracles could be constructed. Different from these studies, our study focuses on investigating bugs that occurred as a result of changes.

Much research has been done to test a single version of Android apps. The behaviors of the tested Android app are explored by using different exploration strategies, including random exploration [5], [15], model-based exploration [6], [19], [8], or systematic exploration [16], [7].

Regression testing has been extensively studied for desktop applications. It is concerned with validating the modified program version after a change. Reusing all of original test suite can be expensive, so various approaches have been developed for re-using tests more effectively by regression test selection [18], [13] and test case prioritization [11], [12], [20].

Regression test selection (RTS) techniques use data on the original and modified program versions, and the original test suite to select a subset of the test suite with which to test the modified program version. One class of RTS techniques, safe techniques (e.g. [18]), guarantee that under certain conditions, test cases not selected could not have exposed faults in the modified program version. Empirical studies have shown that these techniques can be cost-effective. However, due to the difference between desktop platform and Android platform, these approaches cannot be directly applied to mobile apps. Our impact analyzer extends the algorithm in `Dejavu` [18] to cope with Android bytecode to compute the change impact.

Test case prioritization (TCP) techniques reorder the test cases in the original test suite such that testers can more quickly achieve testing objectives, e.g., to reveal the faults in the modified program [12]. Our work is different as it focuses on regression test selection rather than test case prioritization.

VI. Conclusions

In this paper, we presented a bug study based on 10 real-world Android apps from Google Code Repository [3], showing that regressions exist for Android apps during evolution; we also presented a manual case study on change impact across the Android activity tree, which shows that regression testing methods can be potentially made more efficient by focusing on changed sections within an Android application. Motivated by these studies, we introduced `Redroid`, a new approach to regression test selection for Android apps. Given a test suite that was performed on the original Android app, and the two versions involved in a change, `Redroid` identifies a subset of the tests that must be re-executed to test the new Android version.

`Redroid` leverages the combination of static impact analysis and coverage information that is dynamically generated at runtime, and identifies a subset of test cases for re-execution on the modified app version. The execution of those selected test cases can potentially be different from the execution on the original app version. We developed a prototype tool for `Redroid`, and conducted an evaluation based on two real-world Android apps. Experimental results showed that our approach can significantly reduce the number of tests for re-execution, as well as the time cost for re-executing the selected tests after an Android app is modified.

In future work, we plan to conduct a more comprehensive evaluation of our `Redroid` technique. We also plan to have a broader range of static analysis that can be used for change impact analysis on Android platform—not only do we focus on changes of the sources code, but also other kinds of changes, such as library changes and hardware changes.

Acknowledgments

This work is partially supported by the National Science Foundation under Grant No. CNS-1358939.

References

[1] AndStatus. http://andstatus.org/.
[2] EMMA Code Coverage Tool. http://emma.sourceforge.net/.
[3] Google Code Repository. https://code.google.com/.
[4] Inetify. https://code.google.com/p/inetify/.
[5] Monkey. http://developer.android.com/tools/help/monkey.html.
[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *ASE 2012*, pages 258–261, 2012.
[7] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE '12*, pages 59:1–59:11.
[8] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA '13*, pages 641–660.
[9] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru. An empirical analysis of bug reports and bug fixing in open source Android apps. In *CSMR '13*, pages 133–143.
[10] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, Aug. 1997.
[11] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.*, 32(9):733–752, Sept. 2006.
[12] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE '01*, pages 329–338.
[13] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA '01*, pages 312–326.
[14] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *AST '11*, pages 77–83.
[15] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *ESEC/FSE 2013*, pages 224–234, 2013.
[16] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *FSE 2014*, pages 599–609.
[17] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12*, pages 241–251.
[18] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.
[19] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE '13*, pages 250–265.
[20] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ISSTA '09*, pages 201–212.
[21] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *ICST '14*, pages 183–192.