API CHANGE DRIVEN TEST SELECTION FOR ANDROID APPLICATIONS

HONORS THESIS

Presented to the Honors College of
Texas State University
in Partial Fulfillment
of the Requirements

for Graduation in the Honors College

by

Jihan Rouijel

San Marcos, Texas
December 2019

API CHANGE DRIVEN TEST SELECTION FOR ANDROID APPLICATIONS

by

Jihan Rouijel

Thesis Supervisor:

_____

Guowei Yang, Ph.D.
Department of Computer Science

Approved:

_____

Heather C. Galloway, Ph.D.
Dean, Honors College

ABSTRACT

People are becoming increasingly dependent on their mobile devices day after day. As a result, mobile developers strive to introduce innovative applications (or simply apps) with new functionalities and features. To keep up with this need; Android Application Programing Interfaces (APIs) are updated frequently, which can affect the functionality of the Android apps that are built upon them. Developers therefore need to test their apps between each update to ensure high reliability. However, running the entire test suite can be costly and time-consuming. Previous research has focused on how Android API change impacts the app code, but no research has extended this finding to further analyze the API change impact on app tests to help increase the efficiency of API change driven regression testing. This paper introduces a novel approach which leverages the results from app code impact analysis and coverage information to identify the tests that are impacted by the API changes. The results of experiments on real-world android mobile apps show that this approach increases the efficiency of android app testing by reducing the number of tests selected for re-execution and thus reducing the time needed for regression testing.

I. INTRODUCTION

The mobile apps market is one of the fastest growing areas in the information technology [2] and statistically, the *Android* OS is the most widely used mobile operating systems. In 2017, it achieved a major milestone of having over two billion monthly active devices [3]. To keep up with this achievement, developers are required to ensure high quality Android mobile apps.

Regression testing is an expensive maintenance activity aimed at showing that an application has not been adversely affected by changes, estimates indicate that software maintenance activities account for as much as two-thirds of the cost of software production [6]. However, regression testing is necessary after any kind of change that may potentially affect an application. In addition, Application Programming Interfaces (APIs) are frequently updated for Android OS, which from the first commercial release of Android 1.0 with API level 1 on September 23rd, 2008 [20] to the current API level at 29. With these frequent updates, it is essential to keep the app version updated to match the current API version; otherwise the interaction between the apps and the underlying OS may cause some unexpected errors.

Given the complexity of regression testing, the frequency of Android API updates, and the need of developers to keep the apps up-to-date with the current API version; we believe that impact analysis, a process to identify and analyze the effect a change or an update can have on a program [21], plays an important role in achieving high reliability for Android apps. It includes effectively locating the impacted portions of code and determining how much impact the update has.

Multiple testing techniques have been proposed in the recent to improve the reliability of Android apps. However, none of them are specifically targeting on testing the impact of API updates. Moreover, without knowing the impact of API updates, developers must execute all tests on mobile apps with the new API version, which can be very expensive.

In this paper, we introduce a novel tests impact analysis for Android API updates. This approach leverages the results from app code impact analysis introduced by Yang et al. [1] and coverage information for each test in the existing test set to identify the tests that are impacted by the changes in an Android operating system. Only the impacted tests, instead of the whole set of existing tests, need to be re-executed for testing the app after the API changes. As a result, developers could save time in revealing all potential faults caused by an API update that could be revealed with the whole test suite.

To evaluate the effectiveness of our approach, we conducted experiment on 14 real-world Android apps with tests. The results show that the test impact varies widely across different apps and that our approach can effectively improve the efficiency of testing for API updates with reasonable overhead.

III. APPROACH

A. Overview

Given two consecutive API versions V_{old} and V_{new}, and an Android app developed with API V_{old}, our approach aims to identify the subset of tests of the original test suite that were impacted by the API changes. This subset represents the set of tests that cover the impacted app code and that may reveal potential bugs caused by the API changes. Running only these tests will yield the same results as if the whole test suite is executed.

First, our approach analyzes the impact of API changes between API V_{old} and V_{new} on the app; the two API versions need to be consecutive. Second, our approach collects coverage information for each test. Leveraging the impacted app code and test coverage, our approach can identify and select the impacted tests that cover (or execute) the impacted app code.

B. Framework and its implementation

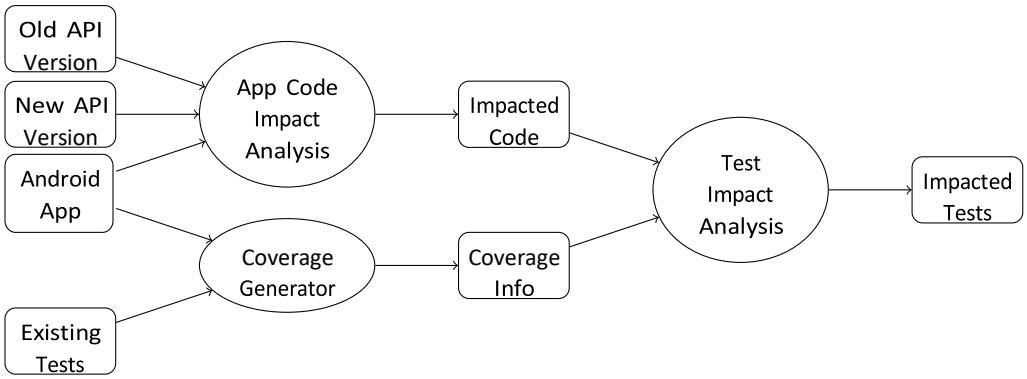

Figure 1 Framework of the approach.

The framework of our approach, as shown in Fig. 1, consists of three major components: App Code Impact Analysis, Coverage Generator, and Test Impact Analysis:

1) App Code Impact Analysis:

The goal of the App Code Impact Analysis is to identify the portion of the app code impacted by the API changes. It takes as input the two consecutive API versions and an Android app's source code. Previous work by Yang et al. [1] introduced an algorithm that automatically locates the part of the app affected by API update. Using this work, we were able to extract a list of affected user methods. First, from the two API versions; the impact analyzer identifies the name of the library methods that were either deleted or modified moving from an API version to a newer one. Then, from

the app's source code, it checks to see if the app's source code methods are using any of the impacted library methods. If a user method is calling any of the impacted library methods, then that user method is marked as impacted.

2) ***Coverage Generator:***

The purpose of the Coverage Generator is to generate a list of covered user methods for each test. For this, it takes as input the app existing tests. First, the coverage generator extracts a list of the names of the test classes and their methods. Then, it uses this list to run each test separately and get its coverage, or the list of user methods that are to be tested by this test.

To extract the list of the names of the test classes and their corresponding methods, we implemented a custom test names generator for Android App using a self-built miniature lexer and parser for Android grammar using Python and regular expression, and then saved the tests Classes and methods names in a file as a dictionary.

Now that we have a list of test names in form of <class>.<method>, we can run each test individually and get its coverage using JaCoCo, a free and easy to use Java code coverage library distributed under the Eclipse Public License [4]. JaCoCo generates a coverage report in several formats including HTML, XML, and CSV. We chose to generate our report in HTML because HTML is more visual which makes it easier for us to scrape the data we need. To extract the covered methods from the coverage report, we used Beautiful Soup, a Python library for pulling data out of HTML and XML files [5]. The covered methods for each test are stored in a

dictionary, with the test being the key and the list of its covered user methods being its value.

3) ***Test Impact Analysis:***

The Test Impact Analysis works to combine the output of the two other components to be able to select the impacted tests. That is, it takes as input the list of the impacted user methods (as determined by the App Code Impact Analysis) and the covered method for each test (as generated by the Coverage Generator).

The Test Impact Analysis loops through each test and checks if any of its covered methods is also in the list of affected methods. If so, the test is considered impacted and selected for re-execution.

In all parts of our framework, we ignore the package level and class level analysis and we only consider method level analysis. The more coarse-grained the analysis is, the less accurate the results will be. This will lead to a greater number of tests being unnecessarily selected.

IV. EVALUATION

A. Research Questions

We will evaluate the effectiveness of our technique by answering our following research questions:

- RQ1: How does an API update impact the app's existing tests?
- RQ2: How does our technique perform in identifying impacted code that is not being covered by any tests?
- RQ3: How does the cost of selecting and executing impacted tests compared to executing all tests?

B. Experimental Setup

For the purpose of this evaluation, we selected 14 real-world Android apps, 12 different apps and 3 different versions of one of the apps, based on some specific criteria as shown in Table 1 and explained as follow:

- Apps are open-source and are available via F-droid (an open-source platform) or GitHub, to be able to get the source code of the app and the original test suite.
- Apps are available on Google Play Store to obtain their statistics.
- Number of downloads are greater than 5000.
- Apps of various categories were chosen to not to be biased against a specific category.
- Apps are recently updated, to make sure that they are regularly maintained.
- Apps written in different and various underlying API to best portray the effectiveness of our approach.

Table 1 Some of the Applications' criteria

App Name	Downloads	Category	Date Last Updated
Kouchat	5,000+	Communication	August 7, 2018
AnkiDroid	5,000,000+	Education	October 15, 2019
gnucash	100,000+	Finance	June 27, 2018
Materialistic	100,000+	News and Magazines	March 30, 2019
AmazeFileManager	1,000,000+	Tools	September 13, 2019
Wikipedia	10,000,000+	Books and Reference	October 11, 2019
NewPipe	10,000+	Health and Fitness	August 27, 2017
Connectbot	1,000,000+	Communication	November 9, 2018
Kdeconnect	500,000+	Productivity	September 23, 2019
Telecine	800,000+	Tools	December 18, 2018

Our experiments are performed on a PC with a i5 Quad-Core 2.50 GHz processor, 8 GB of RAM, and running Windows 10 64-bit operating system.

To build the Android apps and run tests for the apps, we used the following tools:

- Java SE Development Kit 8
- Android Software Development Kit 29.0.1
- Android Studio 3.5
- Gradle 5.6

C. Results

Table 2 Experimental Results

App Name	API Update	Tests			Time (s)			
		Total	*Impacted*	*% Impacted*	*Re-run All Tests*	*Re-run Selected Tests*	*Saved*	*Overhead*
Materialistic	26 → 27	444	302	68.02	9,886.31	7,200.79	2,685.52	537.85
Materialistic	27 → 28	436	257	58.94	10,189.13	6,813.90	3,375.23	563.58
Materialistic	28 → 29	312	283	90.71	8,755.22	8,155.65	599.57	451.71
Kouchat	27 → 28	550	291	52.91	4,997.63	2,804.03	2,193.60	394.59
AnkiDroid	27 → 28	28	18	64.29	864.50	623.29	241.21	40.72
Gnucash	27 → 28	141	117	82.98	4,637.27	4,086.25	551.02	241.69
Telecine	25 → 26	18	0	0	775.65	0	775.65	42.71
NewPipe	26 → 27	29	11	37.93	765.14	397.34	367.8	42.87
AmazeFileManager	27 → 28	153	57	37.25	5,846.13	2,405.97	3,440.16	395.74
Connectbot	26 → 27	36	24	66.67	1,030.91	710.63	320.28	45.17
Wikipedia	28 → 29	396	326	82.32	15,212.69	13,168.60	2,044.09	1034.71
Kdeconnect	28 → 29	9	9	100	397.6	397.6	0	48.19
FileManager	28 → 29	20	1	5	869.06	43.56	825.5	52.83
CineLog	25 → 26	145	10	6.90	2,664.82	286.71	2,378.11	102.23

Table 2 shows the results of our experiment. It contains the two API versions V_{old} → V_{new}, total number of tests (*Total*), the number and the percentage of impacted tests (*Impacted* and *% Impacted*, respectively), total time needed to re-run all the unit tests after the API update (*Re-run All Tests*), time to run the tests our framework selected (*Re-run Selected Tests*), time we saved by running only the selected tests (*Saved*), and the overhead introduced by running our framework *(overhead)*.

D. Analysis

Based on analyzing the results of our experiment, we can answer the three research questions as follow:

RQ1: How does an API update impact the app's existing tests?

Our results show that every app's test suite is affected differently. The percentage of impacted tests varies from an app to another. Some apps, like *Telecine*, may not be affected at all; others, like *Kdeconnect*, can be 100% affected; and other apps can be partially affected. Investigating this further reveals few factors that can affect how the API update impacts the app's tests. One factor is the amount of impacted methods being covered by the tests. If only rarely covered methods are impacted, then the number of the impacted test will be low. On the other hand, the number of impacted tests becomes high if some of the mostly covered methods are impacted by the API update. *Materialistic* (28 →29), for example, has an impacted utility function called isOnWifi() which is covered by most of the tests which makes the percentage of the affected tests high (90.71%). *Telecine* has no impacted tests while *Kdeconnect* tests were all impacted. Both apps have a very small number of tests which cover either impacted methods (like in *Kdeconnect* case) or unimpacted methods (like on *Telecine* case) only.

RQ2: How does our technique perform in identifying impacted code that is not being covered by any tests?

According to Table 2, *Telecine* has 0% impacted tests and *Kdeconnect* has 100%. As mentioned before, both apps have a very small number of tests; *Telecine* has 18 and *Kdeconnect* has only 9. *Kdeconnect* has 1166 methods 391 of which are impacted. It is

not feasible that only these 9 tests can cover all the impacted methods. Also, *Telecine* has 95 methods; 19 of which are impacted, but none of them are covered by any of the 18 tests. For both cases, most impacted methods are not covered by any test and thus will remain untested. However, our approach can easily identify the impacted methods that are not covered by the existing tests. We can find the impacted methods that are not covered by any existing test by subtracting covered methods (generated by our Coverage generator) from the set of impacted methods (outputted by the App Code Impact Analysis.) These results can further help developers augment tests to cover these uncover impacted methods.

RQ3: How does the cost of selecting and executing impacted tests compared to executing all tests?

Results from Table 2 show that our approach was successful in saving time in retesting the apps after an API update. For almost all the apps we use for our experiment, our approach saved a considerable amount of time compared to re-executing all tests. Our approach saved 3,449.16s, for retesting *AmazeFileManager*, 3,375.23s for *Materialistic* (API update 27→28), and 2,685.52s for *Materialistic* (API update 26→27).

The only case where the Overhead time is greater than the Saved time is *Kdeconnect*. As mentioned before, *Kdeconnect*'s tests were not well designed, and coverage of the tests is only 2% (according to JaCoCo test report).

The time saved is also different from one app to another. Tracing the percentage of the time saved by our approach and the percentage of the API test impact in one graph,

as shown in figure 2, shows that these two metrics are related. As the API test impact increases, the time saved decreases; and vice versa.

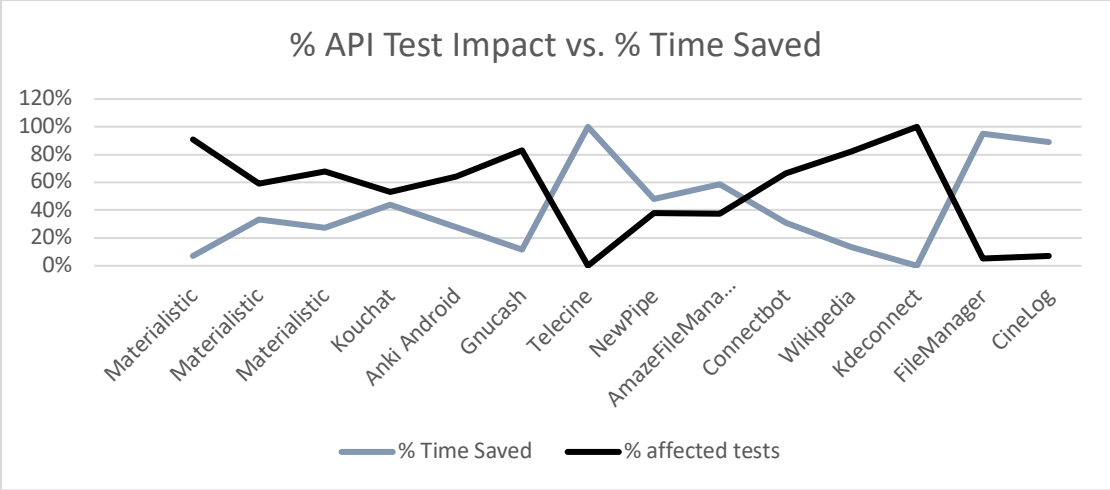

Figure 2: % API test impact vs. % time saved

V. RELATED WORK

Regression testing has been an ongoing research area in software engineering field. Traditionally, the entire test suite has been reused for retesting an app after a change, which can be computationally expensive. Therefore, different techniques have been developed to increase the efficiency of this testing by reducing the number of test cases, which are known as regression test selection [11], [12], [13] and test case prioritization [14], [15], [16]. Do et al. [17] proposed a Regression Test Selection approach for Android apps, Redroid. Redroid leverages the combination of static impact analysis and dynamic code coverage, and identifies a subset of test cases for re-execution on the modified version of Android apps. Li et al. [18] proposed ATOM, for automatic maintenance of GUI test scripts for evolving mobile apps. ATOM uses an event sequence model (ESM) to abstract possible event sequences in a GUI and a delta ESM to abstract the changes made to a GUI, and updates the test scripts that were written for the base

version app, based on the ESM for the base version and the delta ESM for the changes introduced by the new version. Aggarwal et al. [19] proposed GreenAdvisor, a software evolution impact analyzer on energy consumption. It compares the system call logs two consecutive versions of Android apps and predicts how the energy-consumption profile of the new version will compare to that of the previous version.

Impact analysis has also been studied in the past few years. Ryder et al. [7] and Xiao-Bo et al. [8] focused on change impact analysis for object-oriented programs. Based on this work, Ren et al. [9] proposed a tool called Chianti for java programs. Chianti analyzes two versions of a java program and decomposes their difference into a set of atomic changes. Zhang et al. [10] introduced a change impact and regression fault analysis tool for evolving java programs. While these works focus on the impact of changes in Android apps functionality and energy consumption of android application, our work is different as it focuses on the impact of changes in Android operating system on Android apps tests and how this helps us increase the efficiency of selection testing.

VI. CONCLUSION

Regression testing is a complex and expensive part of the application maintenance process. In addition, Android APIs are updated frequently which leads to a more frequent need for retesting applications. In this paper, we introduced a novel approach to analyzing API change impact analysis on application tests. We introduced a framework that is capable of identifying the application tests that are impacted by API changes and selecting these tests for re-execution when APIs get changed. We performed experiments on 14 real-world Android apps to demonstrate the usefulness and the effectiveness of our approach. The experimental results showed that our approach can effectively select tests

that are impacted by API changes and reduce the number of tests for many apps, which improves the efficiency of testing apps with changed APIs.

REFERENCES

[1] G. Yang, J. Jones, A. Moninger, and M. Che. How do android operatingsystem updates impact apps*?* *InProceedings of the 5th InternationalConference on Mobile Software Engineering and Systems,* MOBILESoft'18, pages 156–160, New York, NY, USA, 2018. ACM.

[2] G. Bavota, M. Linares-Vasquez, C. Bernal-C ´ ardenas, M. Di Penta, ´ R. Oliveto, and D. Poshyvanyk. *The impact of api change and fault-proneness on the user ratings of android apps*. IEEE Transactions on Software Engineering (TSE), 2015.

[3] B. Popper. *Google announces over 2 billion monthly active devices on android,* 2017.

[4] JaCoCo. https://www.eclemma.org/jacoco/

[5] BeautifulSoup. https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[6] G. Rothermel, M.Harrold. *Analyzing regression test selection techniques*.IEEE Transactions on Software Engineering, 22(8):529–551, August 1996.

[7] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop*

[8] Z. Xiao-Bo, J. Ying, and W. Hai-Tao. Method on change impact analysis for object-oriented program. In *2011 4th International Conference on Intelligent Networks and Intelligent Systems*, pages 161–164, Nov 2011.

[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[10] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 40:1–40:4, New York, NY, USA, 2012. ACM.

[11] Q. Do, G. Yang, M. Che, D. Hui, and J. Ridgeway. Regression test selection for android applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 27–28, New York, NY, USA, 2016. ACM.

[12] 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 *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.

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

[14] Hyunsook Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, Sep. 2006.

[15] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 329–338, May 2001.

[16] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the*

Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09, pages 201–212, New York, NY, USA, 2009. ACM.

[17] Q. C. D. Do, G. Yang, M. Che, D. Hui, and J. Ridgeway. Redroid: A regression test selection approach for android applications. In *The 28th International Conference on Software Engineering and Knowledge Engineering, SEKE 2016, Redwood City, San Francisco Bay, USA, July 1-3, 2016*., pages 486–491, 2016.

[18] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li. ATOM: automatic maintenance of GUI test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 161–171, 2017.

[19] K. Aggarwal, A. Hindle, and E. Stroulia. Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 311–320, 2015.

[20] Wikipedia. Android version history, 2018.

[21] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.