The Journal of Systems & Software 200 (2023) 111664

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Analyzing the impact of API changes on Android apps☆

Tarek Mahmud [a], Meiru Che [b], Guowei Yang [c,*]

[a] Texas State University, San Marcos, TX, USA
[b] Data61, CSIRO, Brisbane, Australia
[c] The University of Queensland, Brisbane, Australia

ARTICLE INFO

Article history:
Received 2 September 2022
Received in revised form 9 December 2022
Accepted 23 February 2023
Available online 27 February 2023

Dataset link: https://github.com/TSUMahmud/APICIA

Keywords:
Android
API evolution
Change impact analysis
Regression testing

ABSTRACT

The continuous evolution of Android mobile operating system leads to regular updates to its APIs, which may compromise the functionality of Android apps. Given the high frequency of Android API updates, analyzing the impact of API changes is vital to ensure the high reliability of Android apps. This paper introduces APICIA, a novel approach to analyzing the impact of API changes on Android apps. APICIA investigates the impact of changing the target API and identifies the affected program elements (i.e., classes, methods, and statements), the affected tests whose executions may exhibit changed behaviors as a result of the API update, as well as the app code that is not covered by the existing tests. We evaluate APICIA on 219 real-world Android apps. According to the results, API changes impact 46.30% of tests per app on average, and regression test selection based on APICIA can be cost effective. Moreover, many affected statements are not covered by existing tests, which indicates APICIA can help with test suite augmentation to achieve better coverage. These findings suggest that APICIA is a promising approach for assisting Android developers with understanding, testing, and debugging Android apps that are subject to rapid API updates.

1. Introduction

Mobile devices, such as smart phones and tablet computers, have become increasingly useful in various areas, such as study, entertainment, social networking, and business. According to Google Play Store (Google Play Store, 2012), the number of available apps exceeded 2.6 million in 2018 (Statista, 2019). Android is the most commonly used mobile operating system, accounting for more than 70% of the market share (Mobile, 2020). On one hand, developers use Android application development framework to develop Android apps, and they leverage the application programming interfaces (APIs) in the framework to access Android stack functionality on mobile devices. On the other hand, developers are facing the challenges caused by the frequent update of APIs. From the first commercial release of Android 1.0 with API level 1 on September 23, 2008 (Android Version History, 2011) to the present API level 31, Android APIs have frequently evolved. This rapid evolution of APIs causes various compatibility issues for consumers using Android apps (Wei et al., 2016).

Android developers tend to use the latest APIs in order to unlock new features and prevent apps from crashing in updated Android versions due to changes in system behaviors. However, it is non-trivial to update the target API level. According to the Android API Migration guide (Migrating, 2018), to identify the places where the program may be affected, developers should analyze the system behavior changes in the API level they wish to update. Then they need to execute tests focusing on the used API's system behavior changes and for that they need to check the whole app's workflows. When issues are found, they need to locate the API usages that are responsible for the unexpected behaviors, and make changes in the code to fix these issues. In the absence of automated impact analysis, developers may have to manually identify the affected elements in the code and run all the tests to uncover the unexpected behaviors caused by the API level update.

Several approaches for detecting API-related compatibility issues have recently been proposed (Wei et al., 2016; Li et al., 2018; Huang et al., 2018; Wei et al., 2019; Mahmud et al., 2022,b). However, they suffer from the imprecision of static analysis and hence return numerous false positives. Some testing methodologies (Coppola et al., 2017; Ami et al., 2018; Machiry et al., 2013; Mahmood et al., 2014; Zhang, 2018) also have been presented to improve the reliability of JAVA or Android programs by modifying the app source code or byte code, but none of them focuses on evaluating the impact when APIs are evolved and affect the source code. Without knowing the impact of API updates, developers have to re-run all existing tests on Android apps with the

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: tarek_mahmud@txstate.edu (T. Mahmud), meiru.che@data61.csiro.au (M. Che), guowei.yang@uq.edu.au (G. Yang).

https://doi.org/10.1016/j.jss.2023.111664
0164-1212/

new target API level, which can be expensive yet may not discover all compatibility issues if the affected app code is not well covered by the existing tests. Therefore, we believe that impact analysis, a process to identify and analyze the effect a change or an update can have on a program (Arnold and Bohner, 1996), plays an important role in achieving high reliability for Android apps. Previous studies have shown that small changes can have major and non-local effects in object-oriented programs (Ryder and Tip, 2001; Xiao-Bo et al., 2011; Ren et al., 2005). However, to the best of our knowledge, there is no effective technique to analyze the impact of Android API updates on app's source code and tests.

To address this problem, this article presents APICIA a novel approach to analyzing the Impact of API Changes on Android Apps. The API change impact is reported by APICIA in terms of affected program segments (i.e., classes, methods, and statements), as well as affected tests whose executions may exhibit altered behaviors as a result of the API update. Given an API update involving two API levels, APICIA carries the following steps: examining the API differences between two API levels and summarizing them into a collection of API changes; analyzing the app's source code along with API changes to determine the affected app code; collecting coverage information for each test to identify affected tests; and identifying affected source code that is untested by existing tests. The purpose of APICIA is to help Android app developers with (i) identifying API changes and app code affected by the target API change, (ii) reducing the time and effort needed to run regression tests by only choosing tests that are affected by API change, (iii) effectively debugging by locating the API usages in the app code that are responsible for compatibility issues discovered in testing, and (iv) directed test suite augmentation by focusing on affected app code that has not been tested.

To summarize, this paper makes the following contributions:

- We present APICIA, a novel technique for analyzing the impact of API change for Android apps. To the best of our knowledge, this is the first impact analysis on API change for Android apps.
- We discuss the effectiveness of APICIA, on better understanding the impact of API change on app code, on improving test selection, and on easier debugging.
- We evaluate APICIA on 219 real-world Android apps with more than 100 tests. The results demonstrate that APICIA successfully identifies the app code and tests that are impacted by the target API change. Additionally, it lists numerous affected statements in the source code that have not been tested by the existing tests, which helps developers effectively on test suite augmentation.

This article is an extended version of our preliminary work presented at COMPSAC 2021 (Mahmud et al., 2021), where we considered only regular API methods in the API updates. In this article, we redesign APICIA to analyze the impact of changes in regular API methods, API fields, as well as overridden API methods, and we conduct a more comprehensive evaluation of APICIA on 219 real-world apps with more than 100 tests in each apps.

2. Background

This section introduces Android OS, API and API updates, followed with discussion on what developers has to do when they update the target API level of Android apps, which serves as the foundation of our technique.

2.1. Android OS and API

Android OS was introduced in 2008 and is primarily intended for touchscreen mobile devices, such as smartphones and tablets. It was created by the Open Handset Alliance, a business alliance for developing open mobile device standards.

The Android Software Development Kit (SDK) is a set of tools that enables developers to develop apps for the Android operating system. It includes libraries needed to build Android apps, as well as a debugger, emulator, APIs, and sample projects with source code, so developers have everything they need to get started with developing apps. APIs are used to communicate with the underlying Android system, allowing developers to access capabilities such as cameras, location services, wi-fi connections, storages, and so on.

Android is continuously updated in order to accommodate new feature requests, address issues, new standards, and performance improvement. Since its first release, more than 60 versions of Android OS have been launched. In the meanwhile, Android APIs have been developed to help developers leverage the functionalities in the Android OS, and there have been a total of 31 API levels released to date.

2.2. API updates

API updates ensure that the new API level remains compatible with previous API levels, which means that the majority of changes introduced in the new API level are additive with newly added or modified functionalities. The older replaced APIs are deprecated rather than removed, so that existing apps can continue to utilize them. Some APIs may be updated or withdrawn, but these changes are normally only essential to ensure the API's robustness and the application's safety. All other APIs from previous levels are carried over unchanged.

As many Android devices receive over-the-air system updates, forward compatibility is vital. Users may install and use an app before receiving an update to a new version of the Android operating system. After installing the update, the app operates in a new run-time environment that includes APIs and system capabilities which the application depends on. When the app runs in the new environment, it may be affected by API changes in some scenarios. As a result, it is critical for a developer to understand how the app will behave in each run-time environment Forward compatibility (2014).

Google strongly advises developers to specify the minSdkVersion, targetSdkVersion, and maxSdkVersion properties in the manifest or Gradle file (Forward compatibility, 2014). Developers define the range of API levels using the minSdkVersion and maxSdkVersion parameters accordingly. The targetSdkVersion indicates the API level on which an app is intended to run. Targeting a specific API level enables the app developer to use all of the system runtime behaviors corresponding to that API level. On the contrary, an app may crash if it is not prepared to support the new runtime behavior changes introduced in an API level.

If an app is designed for a lower API level, it may not utilize the most recent Android OS functionalities properly. This is why developers should aim for the latest API level to develop high-quality apps with cutting-edge technologies. Google is also enforcing apps to satisfy new target API level requirements, so it is a high priority for developers to keep this value updated (Google, 2022).

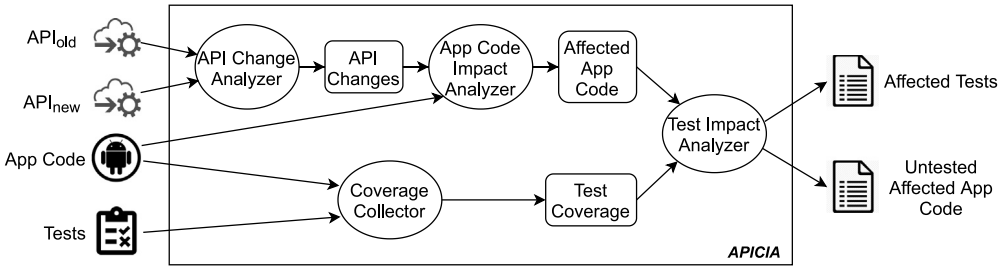


Fig. 1. APICIA overview.

2.3. Updating the target API level

The API level update is unique from other source code modifications. Developers can change the targetSdkVersion in the Gradle file to update it. However, it cannot guarantee a successful API update. To ensure a successful update, developers should still test the app. According to the Android API migration guide (Migrating, 2018) and compatibility testing for Android 11 guide (Test, 2020), developers should review the system behavior changes in the API version they intend to update to identify the areas where the app may be affected, as these changes may cause app crash. They also need to run tests focusing on the app functionalities that have utilized the system behavior modifications. The corresponding code changes finally need to be done in order to address issues after finding unexpected behaviors.

Although Android developers need to specify the target API level, they are not bound to use the APIs available at that specific API level. Usually, Android developers can use APIs available at API levels other than the target API level to use different features in their apps which are available in different API levels. That is why the Android compiler (Android Studio) does not detect them as errors. However, some of these APIs can cause compatibility issues when we run the app on some Android devices with different API levels. To determine which APIs can cause issues, the developers need to test the compatibility of the APIs for all available API levels.

Moreover, traditional code dependency analysis (Li et al., 2013; Egyed, 2003) analyzes code that is dependent on the code changes. However, API changes are not made directly in the app source code, so these changes cannot be analyzed using traditional code dependency analysis. If they are not compatible with the API level of the devices where the app is run, they will cause issues. So, the API change impact analysis is needed to handle these issues.

To the best of our knowledge, there is no impact analysis or regression test selection (RTS) tool available for target API level update. The most advanced RTS tools for Android focus solely on app code changes. As a result, developers must manually identify the areas of an app that are affected by API update, and run all tests to uncover unexpected behaviors induced by the update.

3. Approach

3.1. Overview

Fig. 1 shows the overview of the approach APICIA. APICIA aims to help app developers understand, test, and debug an Android app when updating the app's target API level. Two consecutive API levels are involved in an API update, which are the inputs to APICIA. The source code of an Android app and the tests for the app are another two inputs to APICIA. APICIA indicates the app code affected by changed or removed APIs, determines the tests that are not affected by these API changes, locates the potential API changes and the app code for compatibility issues discovered in testing, and identifies the affected app code that has not been tested by existing tests.

To accomplish this, APICIA first summarizes the differences between two consecutive API versions to identify API changes. Then source code of the app is analyzed to determine which parts of the app code have been affected. After that, APICIA gathers coverage information for each test and utilizes it to identify the affected tests based on the affected app code. Finally, it detects untested affected app code that has been impacted by API changes but has not been tested by existing tests.

3.2. Framework

As shown in Fig. 1, APICIA consists of four components: API Change Analyzer, App Code Impact Analyzer, Coverage Collector, and Test Impact Analyzer.

3.2.1. API Change Analyzer

API Change Analyzer accepts the two API levels involved in an API update as input and returns a list of API methods that are changed or removed during the API update. We note that the previous and new API levels must be successive; for example, if the program was created for API level 27, the new API level should be 28.

APICIA leverages the API differences report to determine all changes involved in an API update. Google publishes an API differences report describing the changes in the core Android framework API between two API level specifications whenever a new level of Android API is released. It shows added, changed, and removed packages, classes, methods, and fields. For example, the summary of changes made from API 27 to API 28 can be found online at https://developer.android.com/sdk/api_diff/28/changes.

To obtain a corresponding report, we can change the number "28" in the URL to any target API level. The report's first page includes three tables: removed packages, added packages, and changed packages. The removed packages and added packages tables list the packages that were removed and added, respectively. The changed packages table lists all of the packages that were changed during the API update. After that, there is a class-level report for each of these packages that includes removed, added, and modified classes. Similarly, in the class-level report, there is a method and field related info for each class. For example, the API differences report for level 27 to level 28 reports the changes to class `android.location.GnssMeasurement` as shown in Fig. 2.

All of the packages that were deleted from the old API are shown in the removed packages table. This table, unlike the changed and added packages table, lacks clickable links because all of the classes, methods, and fields associated with these packages were likewise removed. The table of added packages is disregarded since the added packages, with all of their classes, methods and fields, were not used by the developer in the app and hence had no effect on it. APICIA uses the Android API differences report to produce a dictionary of library classes, methods

Class android.location.GnssMeasurement

Changed Methods	
long getCarrierCycles()	Now deprecated.
double getCarrierPhase()	Now deprecated.
double getCarrierPhaseUncertainty()	Now deprecated.
boolean hasCarrierCycles()	Now deprecated.
boolean hasCarrierPhase()	Now deprecated.
boolean hasCarrierPhaseUncertainty()	Now deprecated.

Added Fields	
int ADR_STATE_HALF_CYCLE_REPORTED	
int ADR_STATE_HALF_CYCLE_RESOLVED	

Fig. 2. Android API differences report for class android.location.GnssMeasurement.

and fields that have been changed or removed as a result of an API update.

The differences report also includes indirectly changed API methods, which we can leverage in our approach more efficiently. Assume there are two API methods, API 1 and API 2, and if API 1 uses API 2 in the code and API 2 changes, Google reports both API methods are changed. Otherwise, we must examine the Android Development Framework (ADF) source code to identify the indirectly changed API methods which would be time consuming.

3.2.2. App Code Impact Analyzer

App Code Impact Analyzer takes the API changes collected by API Change Analyzer and an app's source code as input and returns app code (i.e., classes, methods, and statements) that are affected by the API update.

First, APICIA searches for the use of API methods and fields in the app code. We constructed our own miniature lexer and parser for Android grammar to implement this API usage search in the app code. In this parser, we identify each entity of the source code using regular expressions. First we read a source file and remove all the comments and string literals. We remove them to collect the source code where an API can be used. Then we identify the package and class involved in that source code file and also identify the blocks of each classes. After that, we use regular expressions to identify the variables and the methods inside those blocks. Finally for each method, we collect the statements and variables. Finally, we check if any affected methods or fields collected by the API Change Analyzer, are used in those statements or not.

It appears difficult to gather argument types info that were used in the API method call using this parser, APICIA acquire this information via an intra-procedural dataflow analysis of the methods where the API is used. APICIA also uses this to get information about all the variables (including class attributes) used in the method. Then, using the API usages, APICIA filters these API usages and identify the classes, methods, and sentences where the changed or removed APIs are used. We named them as the affected classes, methods, and statements respectively.

APICIA also examines class hierarchies to see whether the classes extend API classes and override any API methods. It collects these method usages as the API usages too.

This dictionary is important for two major purposes. First, we can use this for test impact analysis. Second, we can compare this dictionary with the coverage information of each test and find out the affected app code that is not tested by any test. Therefore, we can utilize this information for test augmentation which is discussed as possible future work in a later section.

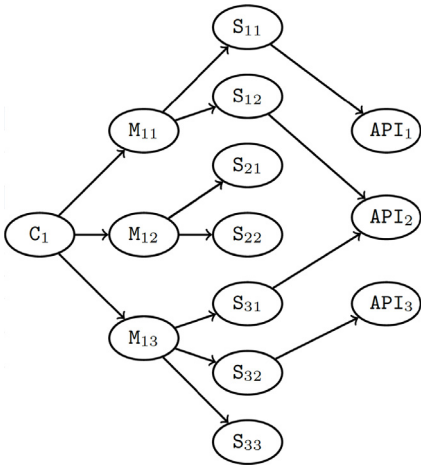


Fig. 3. An example for API method search.

Suppose, we have an Android app that is written using Android API level API_{old}, which will be updated to API_{new}. Using our Android source code parser, we found the App has n classes, C_1, C_2, …., C_n. We specify the methods as M_{nm}, where n indicates the class number this method belongs to and m is the method number. Each method is composed of a number of statements. We specify these statements as S_{mp}, where m indicates the method number this statement belongs to and p is the statement number. These statements use API methods and fields. For example, an app uses q API methods and fields, they are API_1, API_2, …, API_q.

Fig. 3 shows an example class of an app code which we have parsed using our parser. Suppose APICIA identifies API method API_1 and API field API_2 as changed in the API Change Analyzer. As shown in Fig. 3, these APIs are invoked in three statements in the class C_1: S_{11}, S_{12}, and S_{31}. So APICIA marks these three statements, the methods they belong to, and the class they belong to as affected app code. The final results are kept in a nested dictionary as follows : { C_1: {M_{11}: { S_{11},S_{12}}, M_{13}: { S_{31} } } }

3.2.3. Coverage Collector

Coverage Collector takes the app code and the existing tests of the app as input. The purpose of Coverage Collector is to

collect covered classes, methods, and statements for each test. The coverage collector returns a list of covered classes, as well as covered methods and statements in each class.

Suppose, the example app of Fig. 3 has four tests T_1, T_2, T_3, and T_4. T_1 covers S_{11}, T_2 covers S_{12}, T_3 covers S_{21} and S_{22}, and T_4 covers S_{33}. T_1 covers S_{11}, which means that T_1 also covers M_{11} and C_1 since S_{11} is a statement of the method M_{11} and M_{11} belongs to the class C_1. Similarly, T_2 also covers M_{11} and C_1, T_3 also covers M_{12} and C_1, and T_4 also covers M_{13} and C_1. Then APICIA stores the test coverage as following: { T_1: { C_1: { M_{11}: { S_{11}}} }, T_2: { C_1: { M_{11}: { S_{12}}} }, T_3: { C_1: { M_{12}: { S_{21}, S_{22} }} }, T_4: { C_1: { M_{13}: { S_{33}}} } }.

3.2.4. Test Impact Analyzer

Test Impact Analyzer finds the tests that are affected by an API update. It uses the affected app code provided by App Code Impact Analyzer and coverage information for each test provided by Coverage Collector. In particular, for impact analysis at the statement level, the Test Impact Analyzer determines whether any statement covered by a test is affected by the API update and, if so, marks the test as affected. Similarly, for impact analysis at the method-level, it determines whether any affected method is covered by the test under consideration, and if so, the test is considered affected.

In our example from Sections 3.2.2 and 3.2.3, only T_1 and T_2 are selected by the impact analysis because the two affected statements S_{11} and S_{12} are tested by them. T_3 and T_4 are not selected because they are not covering any affected statements. For the method-level analysis; T_1, T_2, and T_4 are selected because they cover affected methods M_{11}, M_{11}, M_{13} respectively.

Since a finer granularity impact analysis is more precise, we use impact analysis with a granularity at the statement level in our work.

3.3. Algorithm and implementation

Algorithm 1 summarizes the implementation of APICIA. It takes the source code `Code` and test suite `T` of an app as input, as well as the two API versions $\texttt{API}_{\texttt{old}}$ and $\texttt{API}_{\texttt{new}}$ that are involved in an API update. It produces the affected tests $\texttt{T}_{\texttt{aff}}$ which is a subset of `T` and untested affected statements unTestedAffectedStatements.

First, APICIA extracts API differences between the app's target API level $\texttt{API}_{\texttt{old}}$ and the new API level $\texttt{API}_{\texttt{new}}$. As the API differences report is only available for two consecutive versions, $\texttt{API}_{\texttt{old}}$ and $\texttt{API}_{\texttt{new}}$ should be consecutive. APICIA stores the changes made in that API update in a hierarchical manner in `changeSummary` (line 8). Then, App Code Impact Analyzer leverages the API method search and the `Code` to create a map of user-defined classes/methods/statements as well as API methods and fields. It generates `affectedClasses`, `affectedMethods`, and `affectedStatements` by comparing class/method/statement maps with the `changeSummary` (line 9–16).

After that, it gathers coverage information for each individual test and saves it in `testCoverageMap` (line 17–18). Then APICIA selects tests `t` whose coverage contains any of the statements in affectedStatements and put it to $\texttt{T}_{\texttt{aff}}$ (line 19–22). It also keeps the covered statements in testedAffectedStatements (line 23). We disregard affectedClasses and affectedMethods from test selection since they are imprecise and do not help us decrease and determine the actual number of affected tests that must be re-executed following the API update. Lastly, it subtracts tested affected statements the testedAffectedStatements from the total affected statement affectedStatements to get untestedAffStatements, which were not covered by the existing tests (line 24).

Algorithm 1: API Change Impact Analysis.

```
   Input: App Source Code Code, Test Suite T, API levels API_old and API_new
   Output: T_aff ⊆ T, untestedAffAppCode
1  affectedClasses := Φ
2  affectedMethods := Φ
3  affectedStatements := Φ
4  testCoverageMap := Φ
5  T_aff := Φ
6  testedAffAppCode := Φ
7  untestedAffAppCode := Φ
   /* API Change Analyzer                                          */
8  changeSummary ← getAPIChangeReport(API_old, API_new)
   /* App Code Impact Analyzer                                     */
9  APIUsages ← extractAPIUsages(Code)
10 for class ∈ APIUsages do
11     for method ∈ class do
12         for statement ∈ method do
13             if statement− > APIMethod ∈ changeSummary then
14                 affectedClasses ∪ {class}
15                 affectedMethods ∪ {method}
16                 affectedStatements ∪ {statement}
   /* Coverage Collector                                           */
17 for t ∈ T do
18     testCoverageMap[t] ← getCoverageInfo(Code)
   /* Test Impact Analyzer                                         */
19 for t, testCoverageInfo ∈ testCoverageMap do
20     for statement ∈ affectedStatements do
21         if statement ∈ testCoverageInfo then
22             T_aff ∪ {t}
23             testedAffAppCode ∪ {statement}
24 untestedAffAppCode ← affectedStatements − testedAffAppCode
```

To obtain the changed API methods and fields in the target API update, APICIA uses Beautiful Soup to scrape the Android API Differences Report webpage. Beautiful Soup is a Python package that extracts data from HTML and XML files. It offers natural ways to navigate, search, and change the parse tree. The above-described webpage is provided by Android, and as mentioned in earlier parts, it covers the changes made between two consecutive API releases. These pages contain packages, classes, and methods that have been removed, added, or changed. We just consider the removed and changed methods and fields in implementing APICIA.

Then APICIA checks whether the affected API methods or fields are used in the app source code or not. To check this, APICIA uses a custom API usage search for Android Apps, for which we implement using a self-built miniature lexer and parser for Android grammar using python and regular expression. As regular expression is insufficient for distinguishing between APIs with similar names and number of parameters, we implement an intra-procedural data flow analyzer to identify the types of the parameters used in the API call. For this, we use `Soot`, which is a Java framework for analyzing, instrumenting, optimizing, and visualizing Java and Android applications (Lam et al., 2011).

To obtain intra- and inter-procedural slices, APICIA takes advantage of Soot's program dependence graph and call graph APIs. In practice, the accuracy of program slicing is determined by the accuracy of statically built call graphs and software dependency graphs. It is well known that statically building precise and sound call graphs and program dependency graphs for Java/Kotlin programs is difficult due to language features such as dynamic method dispatching and reflections. As a result, when examining some apps, APICIA may result in the selection of a few more tests. However, the analysis we used is a conservative analysis, where we select tests to find all the bugs, but not all the tests we select deliver a bug. Like other conservative analysis, APICIA selects tests whose execution may result in finding some compatibility bugs.

Using the result of the API method search, APICIA can identify which class/method/statement is affected by the change by extracting the classes/methods/statements that invoked the changed API methods.

`JaCoCo` (JaCoCo, 2009), a free and easy to use Java code coverage library distributed under the Eclipse Public License, is used for gathering test coverage for each test. The generated classes, methods, and statements are saved in a nested dictionary for further usage.

APICIA retrieves the tests from the app's test report, which can be found in the build folder. We decided not to acquire tests by scanning test files because it is imprecise and may miss some tests if developers create tests in unconventional ways.

4. Evaluation

4.1. Research questions

We evaluate the effectiveness of APICIA by conducting an extensive experiment on real-world Android apps. Our evaluation aims to answer the following five research questions:

- **RQ1: How does APICIA perform in identifying app code that is affected by an API update?**
- **RQ2: How does APICIA perform in identifying tests that are affected by an API update?**
- **RQ3: How does the cost of regression test selection enabled by APICIA compared to re-executing all tests after an API update?**
- **RQ4: How does APICIA perform in identifying affected code that is not covered by existing tests?**
- **RQ5: How do API fields and API methods contribute to the change impact?**

RQ1 aims to study how much affected code can be identified by APICIA in class, method and statement level. RQ2 identifies the tests that are affected by an API update. RQ3 investigates the time saved by running the tests selected by APICIA after an API update. RQ4 studies how effective APICIA can identify the untested affected codes that are not covered by the existing tests. RQ5 aims to investigate how the changes in API fields or API methods impact the app code including classes, methods, statements, as well as the tests during an API update.

4.2. Experimental setup

We selected real-world Android apps based on the following criteria:

- Apps are open-source and are available in F-Droid (2010);
- Apps have a minimum of 100 tests available;
- We should be able to build and run the Apps as well as tests without any errors;
- At least one version of the app is released after the release of Android API 24.

Using these criteria, we selected 219 Android apps in various categories from a total of 2965 apps available in F-Droid which includes popular apps (1M+ downloads) such as Wikipedia, AmazeFileManager, etc. We ensure that the apps undergo routine and timely maintenance, which means developers are continuously and regularly keeping these apps to date. We were able to select apps from 27 categories like Education, Finance, Video Players and Editors, Communication etc., and the selected apps are developed using different types and levels of underlying APIs to best portray the effectiveness of our approach.

Table 1
Affected app code in different granularities.

Granularity	Lowest	Highest	Average
Classes	6.90%	96.6%	71.83%
Methods	2.64%	39.58%	24.16%
Statements	0.33%	2.63%	1.60%

Our experiments were performed on a PC with an i5 Quad-Core 2.50 GHz processor, 8 GB of RAM, and 64-bit Windows 10 operating system. We built each Android app and ran tests for it using the following tools: Java SE Development Kit 8, Android SDK 29.0.1, Android Studio 3.5, and Gradle 5.6.1.

The source code for APICIA as well as the GitHub link of all apps used in the experiments are publicly available for download (Apicia, 2022).

4.3. Results and analysis

4.3.1. RQ1: How does APICIA perform in identifying app code that is affected by an API update?

According to the results, we find that API updates have various impacts on different apps in terms of class-level, method-level and statement-level granularity. Table 1 shows the highest, lowest and the average percentage of source code affected by the API update in different level of granularities. The class-level, method-level and statement-level impacts are respectively 6.90%–96.60% (71.83% on average), 2.64%–39.58% (24.16% on average) and 0.33%–2.63% (1.60% on average). In Figs. 4–6, we can see the distribution of apps in different percentage of affected code fragments. For class-level, the number of apps in the range of 80% to 90% is 56, which is the largest group with the code impact. For method and statement-level code impact, the largest groups are in the range of 20%–25% and 1.2%–1.6% respectively.

Moreover, we notice that the majority of apps have a high class-level and method-level impact. Because of the high impact, if we performed test impact analysis based on class or method-level impact information, we would have to choose and re-run the majority of the tests. As a result, it is logical to focus on statement-level analysis in order to undertake more precise test impact analysis. To summarize, the impact of API changes varies according to code granularity. The impact at the class and method levels is substantial, whereas the influence at the statement level is rather minimal.

4.3.2. RQ2: How does APICIA perform in identifying tests that are affected by an API update?

We conducted a statement-level analysis to study how an API update impacts the existing tests of an Android app. According to the results, on average 46.30% of tests across all apps are impacted by the API update. The highest and lowest percentage of affected tests are 88.78% and 4.14%, respectively. Fig. 7 shows the distribution of the apps according to different percentages of affected tests. We can observe 21.9% apps (the biggest group of apps) have 30% to 40% tests affected by the changes.

By further investigating the results, we find that if API update affects only some rarely used statements, then very few tests are affected. On the other hand, if an API update affects some of the frequently used statements, the number of affected tests becomes large. For example, `Materialistic` has only 2.16% statements affected by the API update; however, a statement as shown in Listing 1, which is in a utility function called `isOnWifi()` and is frequently used, is affected by the API update. As a result, 277 tests out of 312 are selected by APICIA for this app.

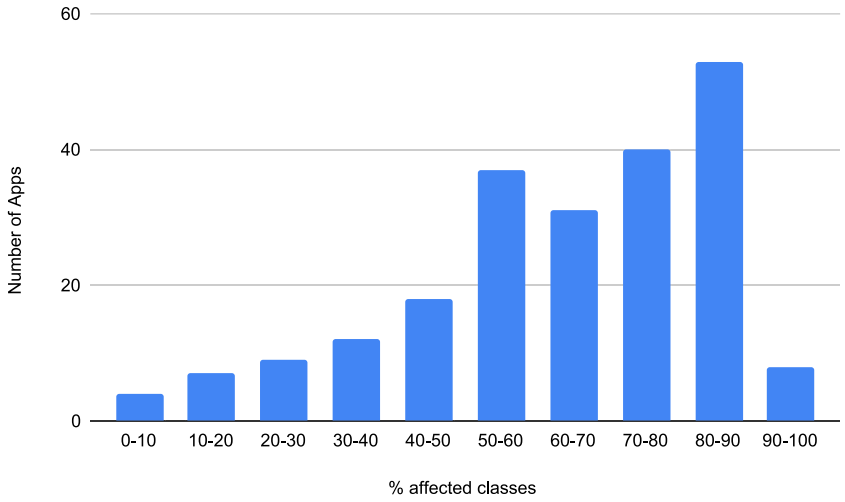


Fig. 4. Number of apps by percentage of affected classes.

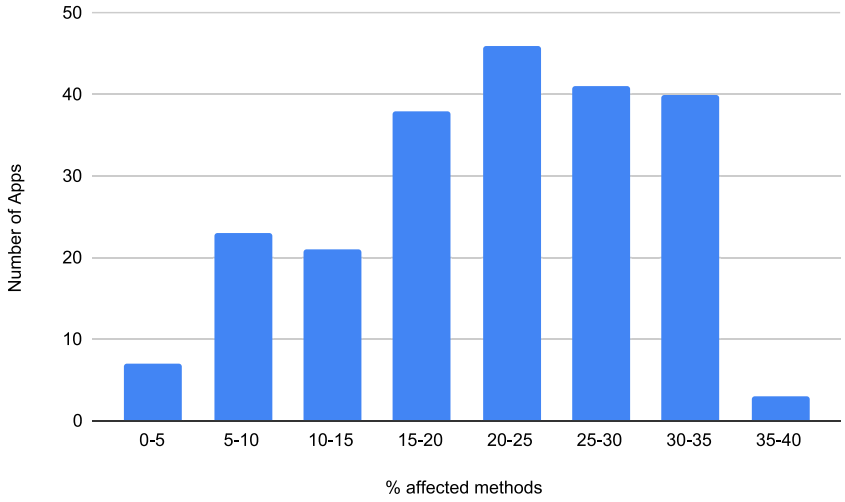


Fig. 5. Number of apps by percentage of affected methods.

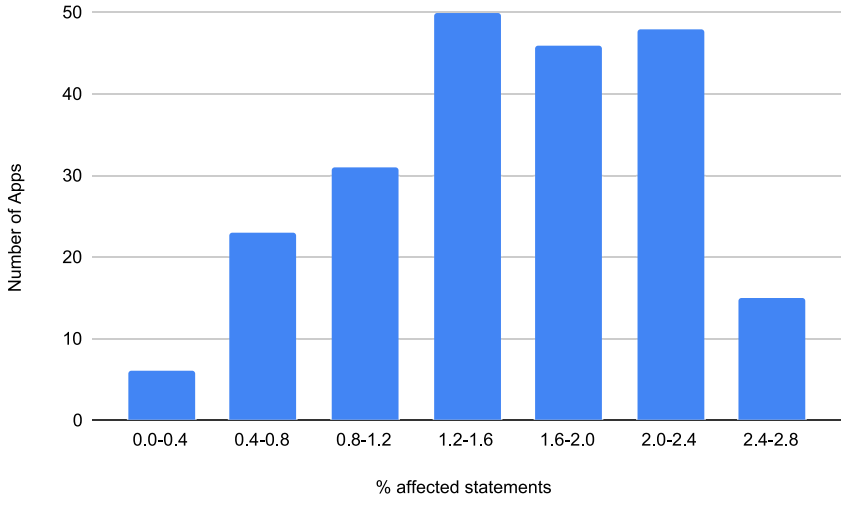


Fig. 6. Number of apps by percentage of affected statements.

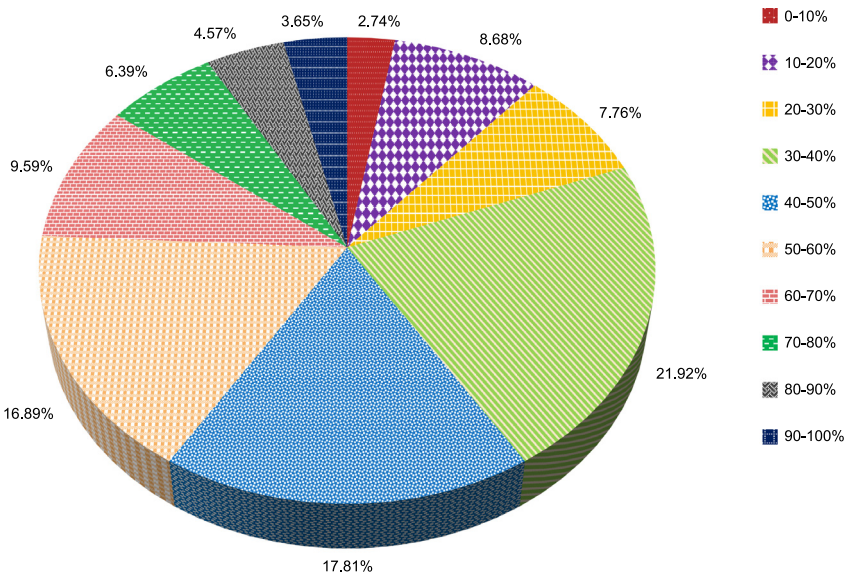


Fig. 7. Distribution of apps according to percentage of affected tests.

```
1 NetworkInfo activeNetwork = ((ConnectivityManager) context
2         .getSystemService(Context.CONNECTIVITY_SERVICE))
3         .getActiveNetworkInfo()
```

Listing 1: An affected statement in Materialistic.

For example, in `Materialistic`, an affected statement shown in Listing 1 is covered by most tests, which indicates this statement could be frequently used, and it is in a utility function called `isOnWifi()`.

To manually evaluate the effectiveness of APICIA in identifying affected tests, we randomly selected 30 apps with 2611 tests. After changing their target API to the next API level, we ran all the tests of those apps. Here, we observed 38 failed tests. On the other hand, APICIA has selected 963 affected tests and all the failed tests are included in the affected tests identified by APICIA. This result demonstrates that APICIA can identify tests that are affected by the API update, allowing for more efficient and effective regression testing.

4.3.3. RQ3: How does the cost of regression test selection enabled by APICIA *compared to re-executing all tests after an API update?*

From the best of our knowledge, there is no tool available for selecting regression tests when updating the target API of an Android app. Also, finding the impacted code for the target API version change is not that straightforward compared to the other type of changes. To uncover unexpected behaviors caused by an API update, developers may need to re-run all existing tests. We compared the cost of running the selected tests by APICIA with the cost of re-executing all tests after an API update to evaluate the cost-effectiveness of regression test selection enabled by APICIA.

We find that, APICIA saved on average 2530.72 s per app, which is a large amount of time. For only 11 apps. APICIA saved less than 1000 s and for 10 apps it saved more than 6000 s. The distribution of the number of apps by the time saved is shown in Fig. 8, where we can see that for most of the apps APICIA saved more than 2000 s.

If we consider the percentage of saved time, APICIA saved more than 80% time for 14 apps. The distribution of the number of apps against the percentage of time saved is shown in Fig. 9. Here we calculate the percentage of time we can save by running only the selected tests instead of re-running all tests. According to Fig. 9, APICIA saved 30% to 70% of time for most of the apps.

We examined the cause of the underperformance for APICIA in some apps. We found out that the test suites in those apps are poorly constructed, with very little test coverage. (We obtained the coverage information by utilizing JaCoCo (2009)) As a result, we can claim that when an app has a well-developed test suite, APICIA can be useful for test selection when the target API level is updated.

Section 3 addressed the case in which the two API levels (old and new) are consecutive. For example, consider an app that was developed with API level 27, which will be updated to API level 28. However, since API level 30 is now available, developers may update the API level from 27 to 30 rather than 28. It is possible for APICIA to handle this by running the `API change analyzer` three times, such that the API changes for API updates 27 to 28, 28 to 29 and 29 to 30 are collected independently and then merged together. According to Algorithm 1, the Line 8 should be $changeSummary \leftarrow getAPIChangeReport(API_{27}, API_{28}) \cup getAPIChangeReport(API_{28}, API_{29})$ to get these API changes. The other parts of the algorithm remain the same.

To evaluate the effectiveness of APICIA in non-consecutive API update, we used the 30 randomly selected apps that we used to manually evaluate the effectiveness in RQ2. The old target API level of those apps ranges from 24 to 29. This time we change their target API level to 32, which is the latest API level available right now. For each app, we have collected the API differences between the old target API level and the API level 32. In this evaluation, the collected class-level, method-level and statement-level impacts are respectively 82.35%, 31.28%, 1.92% on average. And for statement-level analysis, the number of impacted tests are on average 45.76% and out of 2611 test, APICIA select 1195 tests, whose execution led to 61 failed tests. Moreover, execution of only these tests saved on average 1177.22 s per app. These results show APICIA can also be effective for non-consecutive API update.

4.3.4. RQ4: How does APICIA *perform in identifying affected code that is not covered by existing tests?*

APICIA effectively reports the affected statements as well as the affected statements that are not covered by existing tests. According to the results, on average there are 239 untested affected code per app, which is 36.98% of the affected tests. The minimum and maximum number of untested affected code in our experiment are 8 and 935. The distribution of number of apps by

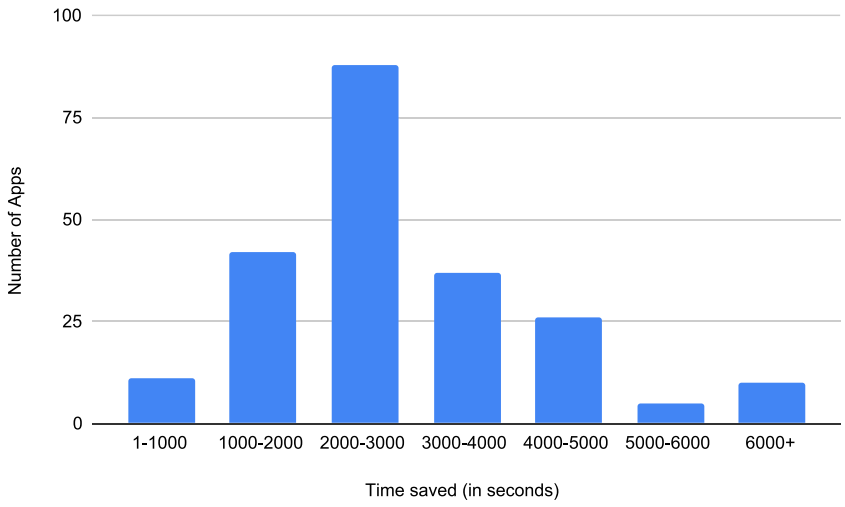


Fig. 8. Number of apps on time saved by running selected tests only.

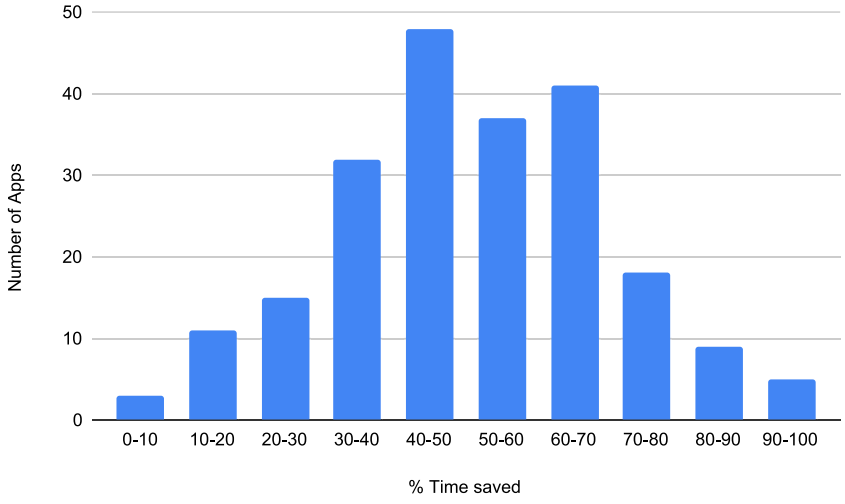


Fig. 9. Number of apps by percentage of time saved.

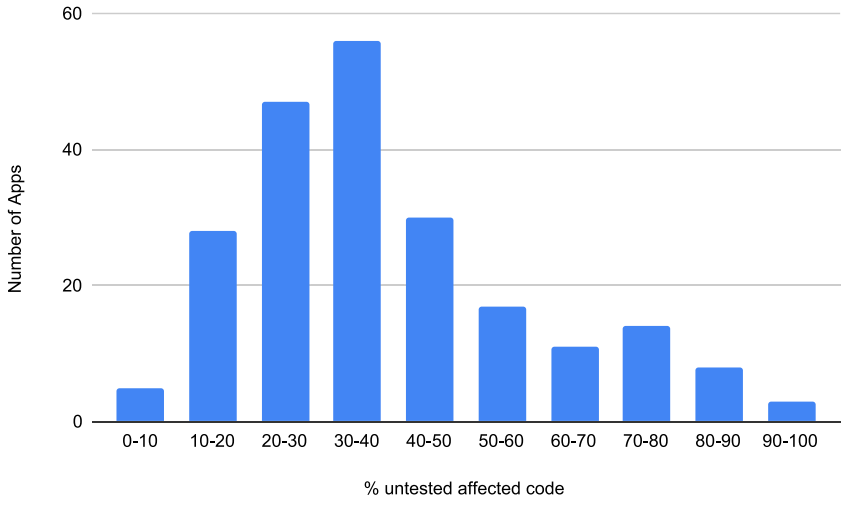


Fig. 10. Number of apps by percentage of untested affected code.

percentage of untested affected code are shown in Fig. 10. The untested affected code span from 2.94% to 94.84% in different apps.

In Fig. 10, we can see that a lot of apps have more than 40% of untested affected code. Also most of the numbers are relatively large. This means that the tests for these apps were not well

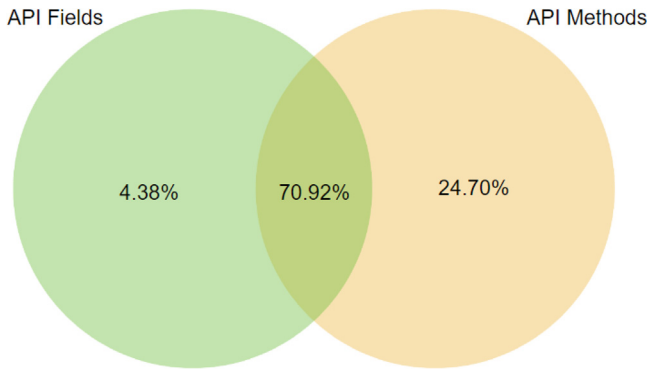


Fig. 11. Affected classes due to changes in API fields vs. API methods.

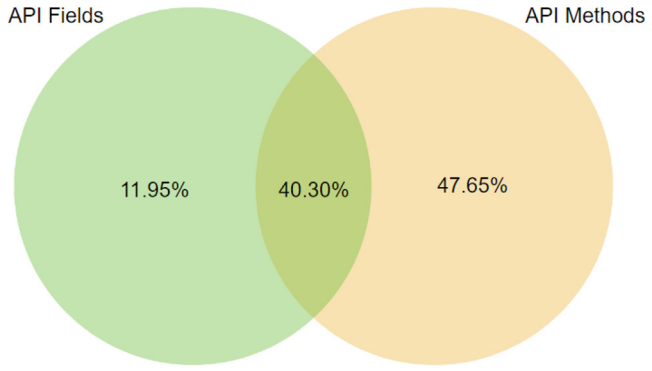


Fig. 12. Affected methods due to changes in API fields vs. API methods.

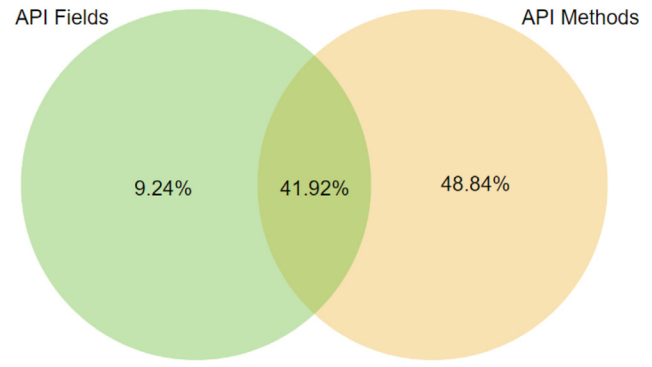


Fig. 13. Affected tests due to changes in API fields vs. API methods.

Table 2
App code affected by changes in API fields vs. API methods.

Granularity	API fields			API methods		
	Lowest	Highest	Average	Lowest	Highest	Average
Classes	2.16%	48.20%	21.43%	5.96%	92.15%	68.68%
Methods	0.95%	14.29%	6.15%	2.33%	37.72%	21.25%
Statements	0.12%	0.99%	0.38%	0.21%	2.35%	1.22%

developed (low coverage) to cover all the API usages, and tests must be augmented to ensure the high reliability of these apps (see Fig. 13).

4.3.5. RQ5: How do API fields and API methods contribute to the change impact?

To answer this research question, we investigated the code in different granularities and the tests that are affected by the changes in two entities of the API, API fields and API methods. Table 2 shows the app code affected by API field changes versus by API method changes. According to the results, 2.16%–48.20% (21.43% on average) classes, 0.95%–14.29% (6.15% on average) methods, and 0.12%–0.99% (0.38% on average) statements are affected by API field changes, while 5.96%–92.15% (68.68% on average) classes, 2.33%–37.72% (21.25% on average) methods, and 0.21%–2.35% (1.22% on average) statements are affected by API method changes. Also, Fig. 11 shows, 4.38% classes are affected only by API field changes, while 24.70% classes are affected only by API method changes. 70.92% classes are affected by both changes. In terms of the impact on method level, the numbers are 11.95% for API fields, 47.65% for API methods and 40.30% for both, as shown in Fig. 12. In terms of the impact at the statement level, 32.84% of statements are affected only by API field changes while 97.16% are affected only by API method changes. No statements were affected by both API field changes and API method changes.

In terms of the impact on tests, on average 22.82% of tests across all apps are affected by API field changes and 41.47% of tests are affected by API method changes. As shown in Fig. 13, 9.24%, 48.84% and 41.92% of the affected tests are affected by API field changes alone, by API method changes alone, and by both types of changes, respectively.

These results suggest that both API field changes and API method changes have impact on app code and tests, while the impact of API method changes is more significant.

4.4. Discussion

We observed that not all updates broke apps while investigating the failing tests. Some API methods are just deprecated because they will be replaced by alternative APIs in the future, and they do not cause apps to fail. Only methods that have changed due to behavioral changes may cause tests to fail. For example, Android 10 makes the changes to classes in the java.util.zip package, which processes ZIP files. Previously, some methods in the Inflater class threw an IllegalStateException if called after a call to end(). In API level 29, these functions instead throw a NullPointerException. If the exceptions are handled by catching IllegalStateException, these changes may cause crashes in Android 10 devices.

In the android app development, APICIA has some other usages too. According to the results in RQ3 and RQ4, APICIA not only can help Android app developers with regression testing by selecting only the tests that need to be re-run, but it also provides insight on test suite augmentation focusing on app code that is affected by the API update but has not been tested by existing tests. This helps developers understand and improve existing app testing, which improves the overall reliability of Android apps.

Statement-level analysis is always more precise than the method-level or class-level analysis. However, statement-level analysis takes more time than the other two, which can be costly for some projects with smaller test suite where the analysis overhead might be higher than the time saved. Although method-level analysis may results in higher number of tests, developers can use it for apps with small test suite instead of statement-level analysis.

4.5. Threats to validity

The selection of applications and API levels utilized to assess our technique, are the primary threats to external validity for our study. The artifacts used in our analysis are all open source apps chosen based on our criteria. The findings of our study may not be applicable to other apps or API versions; however, we mitigated this risk by selecting 219 apps from F-Droid (2010),

where the developers employed different API levels in our evaluation. Additional research on various apps, API levels, and coverage gathering techniques are required to address these concerns.

Potential flaws in the implementation of our algorithm for our approach are the primary threats to internal validity. We mitigated this risk by running our algorithm on samples that we could manually check.

When it comes to threats to construct validity, the metrics we selected are key indicators of the cost effectiveness of impact analysis approaches, but other measures are also possible.

5. Related work

In this section, we discuss the related works on Android API compatibility analysis, regression testing, and change impact analysis.

5.1. Android API compatibility analysis

Wei et al. (2016) investigated the characteristics and core causes of compatibility issues and introduced FicFinder, a static code analysis technique based on a model that captures Android APIs as well as the context in which compatibility issues occur. Li et al. (2018) presented CiD, which identifies compatibility issues using Android Lifecycle Modeling (ALM) and Android Usage Extraction (AUE). ALM examines the source code of the Android framework, whereas AUE generates usage data via conditional callgraphs.

Huang et al. (2018) reviewed official Android documentation and framework code to build a graph that captures control flow inconsistencies, and proposed Cider to detect API callback compatibility issues using the graph. Although it outperforms Lint (Android Lint, 2011), it suffers from the cost of manual effort and, due to its static nature, it produces many false positives.

Wei et al. (2019) introduced PIVOT, which automatically learns API-device correlations of FIC difficulties from previous Android apps. During the learning phase, the technique relies on existing FIC issues, hence it is probable that some of them will be missed. Mahmud et al. (2021a) proposed ACID, which uses Android API differences to detect both API invocation and callback compatibility issues. Compared to previous techniques, Apicia studies how much an API update can affect an app's functionalities including compatibility issues. Like ACID, Apicia also uses API differences, but it focuses on analyzing the impact of an API update in the app code and tests.

AppEvlolve, proposed by Fazzini et al. (2019), automatically updates deprecated APIs in Android apps. By reviewing before- and after-update examples, AppEvolve learns how to update apps that use a deprecated API. These updates include in-code conditional checks for Android versions as well as the use of replacement APIs for the deprecated API.

Haryono et al. (2020) presented CocciEvolve, which uses only one after-update example and normalizes both the after-update example and the target app code before update, but it hampers the readability of the updated code. Then, Haryono et al. (2021) proposed AndroEvolve, which employs data flow analysis and variable denormalization to improve the readability of the updated code. However, these techniques require the developers to know the usages of deprecated APIs, while Apicia analyzes the impact of these deprecated APIs as well as changed APIs on app code and test them, which may lead to the automated discovery of the usages of problematic APIs.

5.2. Regression testing

Regression testing has long been studied with the purpose of testing an updated software version after a modification to ensure that the newly updated version is working fine. Different techniques have been developed to reduce the number of test cases in regression testing, such as regression test selection (Do et al., 2016b; Harrold et al., 2001; Rothermel and Harrold, 1997; Gligoric et al., 2015; Legunsen et al., 2016; Zhang, 2018) and test case prioritization (Yoo et al., 2009; Elbaum et al., 2001; Hyunsook Do and Rothermel, 2006). Unlike these techniques, Apicia focuses on the tests that are influenced by API changes and offers an efficient way to detect potential issues in apps as a result of API changes.

Several approaches for regression testing Android apps have recently been proposed. Do et al. (2016a) proposed Redroid for Regression Test Selection in Android apps. Redroid uses static impact analysis and dynamic code coverage to identify regression tests for re-execution on updated versions of Android apps. ATOM was proposed by Li et al. (2017) for automatic maintenance of GUI test scripts for evolving mobile apps. ATOM abstracts possible event sequences in a GUI using an event sequence model (ESM) and a delta ESM, and updates the test scripts written for the base version app using the ESM for the base version and the delta ESM for the changes introduced by the new version. Other regression testing methods, such as HyRTS (Zhang, 2018), may be adaptable for use on Android apps. However, these regression test selection approaches do not consider the target API level update which is the main focus of Apicia. That is also why we cannot compare the state-of-the-art regression testing tools with Apicia.

5.3. Change impact analysis

Impact analysis has been studied for a long time. Ryder and Tip (2001) and Xiao-Bo et al. (2011) discussed change impact analysis for object oriented programs. Based on this work, Ren et al. (2005) 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. (2012) introduced a change impact and regression fault analysis tool for evolving Java programs. Recently, Yang et al. (2018) studied the impact of Android updates on the apps and provided statistical measures on how much an app is impacted. Our work is different as it focuses on impact of Android API changes.

GreenAdvisor (Aggarwal et al., 2015) is a tool for analyzing the impact of software evolution on energy consumption. It compares the system call logs of 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. While this work focuses on impact of program changes on energy consumption of Android apps, our work focuses on impact of API changes on the functionality of Android apps.

Researchers have studied the evolution of Android APIs. Linares-Vásquez et al. (2013) analyzed the relation between change-proneness and error-proneness API usages with the ratings of Android apps and found out that apps with higher success generally use APIs that are less fault-prone and change-prone than the apps with lower success. McDonnell et al. (2013) conducted an empirical study of the evolution of Android APIs using the version history data found in GitHub, to understand the relationship between API evolution and its adoption. In contrast, our work studies the impact of API changes on app code and tests for the purpose of understanding, testing and debugging for API changes.

Several empirical studies have been conducted by Businge et al. (2012, 2013) and Businge et al. (2015) on the API evolution

of the Eclipse platform. Hou and Yao (2011) studied the Java API documentation specifically the evolution of the AWT/Swing APIs. Wang et al. (2020) presented an exploratory study of deprecated Python library APIs to understand how these deprecated APIs are declared and documented in practice by their maintainers, and how library users react to them.

6. Conclusion

Android OS has become the most widely used mobile operating system. In order to support new Android features in apps, Android APIs are updated frequently and developers join the race to update their apps to support these features. In this paper, we introduced APICIA, a novel approach to analyzing the impact of API changes on Android Apps when their target API is updated. APICIA offers capabilities of identifying app code and tests that are affected by the update, determining a safe approximation of the API changes and app code for debugging, and identifying untested affected code for test augmentation. Our experiments on 219 real-world Android apps demonstrate the utility of our approach, and indicate that APICIA is a promising technique for assisting Android developers with understanding, testing, and debugging for an API update.

CRediT authorship contribution statement

Tarek Mahmud: Conceptualization, Methodology/Study design, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Meiru Che:** Conceptualization, Methodology/Study design, Writing – review & editing, Visualization. **Guowei Yang:** Conceptualization, Methodology/Study design, Validation, Resources, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code of the tool and the information regarding the dataset are publicly available at https://github.com/TSUMahmud/APICIA

Acknowledgments

This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1464123 and CCF-1659807.

Appendix A. Detailed results

Table A.3 shows how much classes, methods and statements in the apps' source code are affected when the target API level is updated to its next level. The column API update shows the current target API and the API it is updating to. The columns #Total, #Aff and %Aff under each program element (including classes, methods and statements) show the total number of elements, the number of affected elements, and the percentage of affected elements, respectively.

Table A.4 shows how much the tests are affected and how much time can be saved by running only the affected tests when the updating the target API level. The columns #Total, #Aff and %Aff under Tests show the total number of tests, the number of affected tests, and the percentage of affected tests, respectively. The Total, Aff and Saved columns under Time indicate the time needed to run all the tests, the time needed to run only the affected tests, and the time saved by only running the affected tests, respectively. The Analysis overhead reports the overhead of APICIA for impact analysis. The table also reports the number of tested and untested affected statements in #Tested and #Untested columns respectively under Affected Code.

Table A.3
Detailed results for affected app code in different granularities.

App name	API update	Classes			Methods			Statements		
		# Total	# Aff	% Aff	# Total	# Aff	% Aff	# Total	# Aff	% Aff
1010!_Klooni	30->31	403	266	66.00%	2284	108	4.73%	13 776	100	0.73%
2_Player_Battle	27->28	512	62	12.11%	3641	555	15.24%	13 231	196	1.48%
A2DP_Volume	26->27	415	353	85.06%	2415	492	20.37%	12 825	270	2.11%
Aard_2	24->25	436	157	36.01%	3652	919	25.16%	20 224	516	2.55%
adbWireless	28->29	107	48	44.86%	709	230	32.44%	2318	18	0.78%
AELF	28->29	427	188	44.03%	2183	815	37.33%	15 659	118	0.75%
AFWall+	28->29	401	373	93.02%	2042	667	32.66%	7256	70	0.96%
Alarmio	26->27	371	279	75.20%	2144	504	23.51%	8842	80	0.90%
ALSA_Mixer_WebUI	25->26	213	49	23.00%	994	246	24.75%	5295	97	1.83%
AmazeFileManager	28->29	236	209	88.56%	1430	566	39.58%	44 737	703	1.57%
Andor_s_Trail	24->25	598	180	30.10%	2827	1091	38.59%	16 761	326	1.94%
AndStatus	28->29	445	263	59.10%	2982	988	33.13%	14 504	196	1.35%
AnkiDroid	30->31	386	170	44.04%	2071	194	9.37%	9021	110	1.22%
Antimine	25->26	537	210	39.11%	2344	434	18.52%	11 991	209	1.74%
AnySoftKeyboard	26->27	414	365	88.16%	2070	714	34.49%	10 350	158	1.53%
ApnSwitch	26->27	318	249	78.30%	2120	473	22.31%	12 840	265	2.06%
ArchWiki_Viewer	26->27	105	78	74.29%	494	82	16.60%	1904	21	1.10%
Audinaut	26->27	486	302	62.14%	2754	176	6.39%	12 646	191	1.51%
Auto_Airplane_Mode	27->28	194	18	9.28%	1067	148	13.87%	6483	37	0.57%
Barnacle_Wifi_Tether	30->31	541	455	84.10%	2509	230	9.17%	11 235	231	2.06%
Beam_File	29->30	507	330	65.09%	2351	390	16.59%	8135	70	0.86%
Better_schedule	24->25	269	226	84.01%	1076	413	38.38%	5131	88	1.72%
Better_Wifi_on_off	30->31	369	270	73.17%	2436	846	34.73%	9440	246	2.61%
Blitzortung_Lightning_Monitor	30->31	484	170	35.12%	3036	221	7.28%	12 034	88	0.73%
BlitzType_Keyboard	27->28	606	43	7.10%	3333	391	11.73%	14 739	62	0.42%
Bluetooth_terminal	26->27	442	76	17.19%	2161	519	24.02%	9668	170	1.76%
Booky_McBookface	30->31	578	428	74.05%	4046	460	11.37%	16 745	345	2.06%

(continued on next page)

Table A.3 (*continued*).

App name	API update	Classes			Methods			Statements		
		# Total	# Aff	% Aff	# Total	# Aff	% Aff	# Total	# Aff	% Aff
Botifier	24->25	598	569	95.15%	4120	415	10.07%	19 227	316	1.64%
Camera	30->31	166	25	15.06%	1328	416	31.33%	8150	184	2.26%
Camp_2019	29->30	111	13	11.71%	478	133	27.82%	2534	55	2.17%
CaptivePortalLogin	25->26	160	127	79.38%	1245	66	5.30%	6194	59	0.95%
Car_Bus_Interface	26->27	315	98	31.11%	2111	72	3.41%	10 782	179	1.66%
CineLog	25->26	116	8	6.90%	417	11	2.64%	9319	31	0.33%
Clementine_Remote	30->31	388	94	24.23%	1764	342	19.39%	10 471	233	2.23%
Clock_Live_Wallpaper	29->30	508	290	57.09%	2083	789	37.88%	6308	152	2.41%
codename_hippopotamos	30->31	296	231	78.04%	1717	89	5.18%	9427	144	1.53%
Color_Namer	27->28	604	478	79.14%	2416	713	29.51%	12 257	118	0.96%
Comfort_Reader	24->25	201	105	52.24%	859	226	26.31%	4510	87	1.93%
Congress_Fahrplan	28->29	583	65	11.15%	2173	534	24.57%	12 182	150	1.23%
Cool_Reader	28->29	286	181	63.29%	2253	261	11.58%	8872	145	1.63%
Countdown_for_DashClock	28->29	543	147	27.07%	2661	913	34.31%	13 851	283	2.04%
Cover_Fetcher	25->26	586	522	89.08%	3297	819	24.84%	13 509	190	1.41%
Crossword	25->26	160	133	83.13%	925	194	20.97%	3269	58	1.77%
Cuberite	25->26	547	186	34.00%	4133	1538	37.21%	17 188	129	0.75%
DashClock__Sunrise	25->26	501	51	10.18%	3229	1106	34.25%	12 629	94	0.74%
Dialer_for_Pebble	29->30	520	162	31.15%	2947	791	26.84%	13 773	346	2.51%
Dicer	27->28	198	46	23.23%	1436	234	16.30%	8421	40	0.48%
DNSSetter	26->27	101	72	71.29%	643	92	14.31%	2853	73	2.56%
DroidFish	29->30	283	114	40.28%	1868	470	25.16%	13 259	150	1.13%
Duorem	26->27	112	70	62.50%	560	157	28.04%	2827	62	2.19%
DuOTP	27->28	226	64	28.32%	1336	112	8.38%	4982	102	2.05%
Easy_xkcd	28->29	424	378	89.15%	1696	154	9.08%	5824	77	1.32%
EasyRSS	26->27	529	403	76.18%	2453	758	30.90%	11 331	129	1.14%
Elementary	30->31	134	21	15.67%	524	162	30.92%	1615	15	0.93%
Enchanted_Fortress	30->31	532	75	14.10%	3606	344	9.54%	14 311	91	0.64%
Equate	28->29	96	64	66.67%	512	75	14.65%	2450	28	1.14%
Equipe_RSS	29->30	617	266	43.11%	3702	143	3.86%	26 231	229	0.87%
Eve-control	24->25	298	69	23.15%	1084	358	33.03%	3671	17	0.46%
Eve-control	28->29	129	99	76.74%	832	116	13.94%	3034	78	2.57%
Exodus	26->27	150	102	68.00%	737	226	30.66%	3738	50	1.34%
EZ_Wifi_Notification	28->29	344	327	95.06%	1376	464	33.72%	4604	56	1.22%
Fabularium	25->26	324	156	48.15%	1620	472	29.14%	6985	65	0.93%
Fast_Brightness_Control_Widget	28->29	622	293	47.11%	3499	851	24.32%	17 037	253	1.49%
Fate_Sheets	27->28	617	309	50.08%	3646	856	23.48%	21 075	116	0.55%
Fire_Chat	24->25	208	86	41.35%	1456	366	25.14%	5163	43	0.83%
Flight_Mode	25->26	206	71	34.47%	1351	112	8.29%	6817	157	2.30%
FOSDEM_Companion	27->28	501	71	14.17%	3207	949	29.59%	16 471	350	2.12%
FOSS_Browser	26->27	257	222	86.38%	1510	249	16.49%	5324	107	2.01%
Freebloks	27->28	272	202	74.26%	1481	286	19.31%	5546	93	1.68%
Giggity	28->29	244	37	15.16%	1287	191	14.84%	5231	55	1.05%
Gilga	28->29	171	91	53.22%	964	91	9.44%	4062	66	1.62%
GM_Dice	28->29	502	141	28.09%	4393	854	19.44%	24 532	182	0.74%
Gnucash	27->28	147	142	96.60%	1364	325	23.83%	36 622	583	1.59%
Great_Freedom	24->25	559	118	21.11%	3479	801	23.02%	16 536	144	0.87%
Hacker_s_Keyboard	25->26	267	171	64.04%	1683	357	21.21%	7645	61	0.80%
Hall_Monitor	28->29	445	250	56.18%	2423	85	3.51%	9093	131	1.44%
Hash_Droid	25->26	443	284	64.11%	2382	689	28.93%	12 334	56	0.45%
Heb12	28->29	442	177	40.05%	3315	538	16.23%	10 521	62	0.59%
I2P	26->27	303	28	9.24%	1482	283	19.10%	7551	84	1.11%
iBeaconDetector	27->28	299	168	56.19%	2093	74	3.54%	13 264	275	2.07%
Inetify	27->28	416	296	71.15%	1589	575	36.19%	5614	99	1.76%
INSTEAD	28->29	221	89	40.27%	1768	122	6.90%	12 398	196	1.58%
J2ME_Loader	25->26	375	75	20.00%	1875	188	10.03%	12 552	269	2.14%
Kakugo	29->30	277	50	18.05%	1262	487	38.59%	6794	74	1.09%
KeePass_NFC	30->31	474	451	95.15%	2543	566	22.26%	7396	126	1.70%
KeePassDroid	28->29	501	451	90.02%	2338	705	30.15%	9478	200	2.11%
Kiwix	24->25	532	91	17.11%	4323	816	18.88%	26 857	691	2.57%
Knock_on_Ports	30->31	128	81	63.28%	664	135	20.33%	3773	67	1.78%
KouChat	30->31	333	210	63.06%	2553	445	17.43%	17 337	172	0.99%
Kouchat	27->28	168	123	73.21%	977	165	16.89%	22 797	245	1.07%
LeMonde	27->28	415	63	15.18%	2646	655	24.75%	14 357	84	0.59%
Lexica	24->25	553	45	8.14%	3318	339	10.22%	12 484	236	1.89%
Logcat_to_UDP	29->30	251	191	76.10%	1883	132	7.01%	10 160	194	1.91%
Look4Sat	28->29	326	304	93.25%	2282	314	13.76%	6760	40	0.59%
Mach3Pendant	29->30	381	355	93.18%	2325	280	12.04%	16 525	305	1.85%
Materialistic	28->29	131	123	93.89%	994	343	34.51%	18 879	408	2.16%
Mensa-Guthaben	30->31	278	34	12.23%	1946	350	17.99%	7695	181	2.35%
microMathematics_Plus	30->31	172	45	26.16%	1101	368	33.42%	5695	59	1.04%
Mines3D	30->31	524	163	31.11%	3406	151	4.43%	16 708	296	1.77%
Minetest	27->28	557	435	78.10%	3899	704	18.06%	18 486	283	1.53%

(*continued on next page*)

Table A.3 (*continued*).

App name	API update	Classes			Methods			Statements		
		# Total	# Aff	% Aff	# Total	# Aff	% Aff	# Total	# Aff	% Aff
MoClock	30->31	545	120	22.02%	3089	1089	35.25%	11 896	49	0.41%
Moonlight	25->26	295	169	57.29%	1564	384	24.55%	10 080	261	2.59%
Morse	25->26	106	14	13.21%	579	194	33.51%	3101	32	1.03%
Moscow_Wi-Fi_autologin	30->31	236	152	64.41%	1076	386	35.87%	6689	85	1.27%
MTG_Familiar	30->31	435	227	52.18%	3094	553	17.87%	17 676	379	2.14%
Music	24->25	486	287	59.05%	3005	185	6.16%	17 450	238	1.36%
Music_Player	26->27	553	471	85.17%	2112	162	7.67%	9915	134	1.35%
Music_Player_GO	24->25	209	84	40.19%	953	159	16.68%	5826	26	0.45%
MusicBrainz	27->28	582	47	8.08%	3069	170	5.54%	19 536	239	1.22%
Muzei_with_Studio_Ghibli	27->28	310	202	65.16%	1891	545	28.82%	6796	66	0.97%
MyOwnNotes	24->25	522	272	52.11%	2819	890	31.57%	10 677	133	1.25%
NCBSinfo	28->29	194	37	19.07%	1014	209	20.61%	4473	70	1.56%
NDEF_Tools_for_Android	29->30	249	152	61.04%	974	191	19.61%	5671	42	0.74%
NewsBlur	28->29	374	315	84.22%	2572	288	11.20%	9517	104	1.09%
NFC_Key	26->27	243	209	86.01%	2096	575	27.43%	12 143	50	0.41%
NFC_Reader	28->29	534	369	69.10%	3525	435	12.34%	18 638	336	1.80%
NFC_Tag_maker	24->25	441	49	11.11%	2045	248	12.13%	10 524	145	1.38%
NFCard	27->28	121	23	19.01%	968	114	11.78%	5571	88	1.58%
NFCMessageBoard	29->30	571	172	30.12%	3541	1082	30.56%	10 589	212	2.00%
NitroShare	26->27	613	313	51.06%	3985	465	11.67%	14 055	317	2.26%
Note_Crypt_Pro	26->27	112	32	28.57%	438	46	10.50%	1769	20	1.13%
Notification_Cron	24->25	438	202	46.12%	2872	281	9.78%	15 181	382	2.52%
Number_Guesser	27->28	204	147	72.06%	928	210	22.63%	4670	43	0.92%
Odyssey	26->27	490	373	76.12%	3158	250	7.92%	18 493	411	2.22%
Offline_Calendar	24->25	400	164	41.00%	2360	190	8.05%	12 356	187	1.51%
OgreSampleBrowser	27->28	347	122	35.16%	1874	146	7.79%	8500	39	0.46%
OI_Notepad	25->26	279	28	10.04%	1860	236	12.69%	9947	59	0.59%
OkcAgent	26->27	203	29	14.29%	970	370	38.14%	4211	50	1.19%
OMW_Nightly	26->27	109	14	12.84%	558	43	7.71%	2227	32	1.44%
Open_Sudoku	29->30	264	154	58.33%	1232	475	38.56%	6314	55	0.87%
Open_WiFi_Cleaner	27->28	352	338	96.02%	1632	440	26.96%	9562	215	2.25%
OpenDNSUpdater	26->27	427	381	89.23%	2468	322	13.05%	10 119	134	1.32%
OpenFoodFacts	28->29	238	164	68.91%	1890	461	24.39%	35 221	659	1.87%
OpenManga	30->31	404	227	56.19%	2335	768	32.89%	16 120	358	2.22%
OpenStud	30->31	228	185	81.14%	1064	187	17.58%	4111	102	2.48%
Oscilloscope	25->26	195	112	57.44%	839	114	13.59%	3263	58	1.78%
PactrackDroid	27->28	270	225	83.33%	1809	311	17.19%	9024	102	1.13%
PassAndroid	30->31	315	249	79.05%	2678	387	14.45%	13 152	89	0.68%
Password_Store	29->30	583	146	25.04%	2173	813	37.41%	14 594	324	2.22%
PDF_Creator	27->28	611	587	96.07%	4507	1538	34.12%	19 463	470	2.41%
Pdf_Viewer_Plus	29->30	457	64	14.00%	2742	1022	37.27%	11 673	236	2.02%
Persian_Calendar	25->26	234	199	85.04%	1463	236	16.13%	4323	53	1.23%
Pixel_Wheels	29->30	176	36	20.45%	1166	205	17.58%	3701	93	2.51%
Planes_Android	26->27	464	251	54.09%	2489	859	34.51%	8351	100	1.20%
Port_Authority	29->30	308	37	12.01%	1771	77	4.35%	9671	101	1.04%
pOT-Droid	29->30	406	94	23.15%	1588	115	7.24%	10 273	158	1.54%
Prayer_Times__Islamic_Tools_	28->29	383	161	42.04%	2979	1015	34.07%	11 402	248	2.18%
PReVo	27->28	446	241	54.04%	2109	126	5.97%	11 073	193	1.74%
Progress_Bars	27->28	362	265	73.20%	2074	313	15.09%	13 616	246	1.81%
Protect_Baby_Monitor	24->25	455	392	86.15%	3287	844	25.68%	9228	167	1.81%
PySolFC	25->26	285	146	51.23%	1296	499	38.50%	6266	35	0.56%
QR_Scanner	25->26	551	359	65.15%	3747	1265	33.76%	22 128	434	1.96%
QRStream	30->31	473	417	88.16%	3010	825	27.41%	17 081	421	2.46%
QuickDic	24->25	238	89	37.39%	1167	390	33.42%	6731	176	2.61%
RadioBeacon	24->25	531	282	53.11%	2602	586	22.52%	13 754	234	1.70%
RadioDroid	27->28	422	216	51.18%	2490	679	27.27%	12 675	230	1.81%
Reader_for_Selfoss	30->31	183	176	96.17%	1121	168	14.99%	4836	30	0.62%
RedReader	25->26	231	93	40.26%	1993	491	24.64%	10 591	50	0.47%
Remote_Droid	25->26	462	79	17.10%	1680	381	22.68%	6254	141	2.25%
RingyDingyDingy	25->26	334	141	42.22%	2714	848	31.25%	10 654	268	2.52%
ScreenStream	26->27	213	39	18.31%	1278	357	27.93%	5565	72	1.29%
SDB_Viewer	26->27	552	194	35.14%	3644	1241	34.06%	24 492	255	1.04%
SealNote	24->25	383	158	41.25%	2452	148	6.04%	11 310	185	1.64%
Seeks	25->26	448	408	91.07%	2159	474	21.95%	11 415	276	2.42%
Sensors_Sandbox	27->28	381	206	54.07%	3048	831	27.26%	21 373	300	1.40%
Shattered_Pixel_Dungeon	29->30	516	253	49.03%	2205	835	37.87%	8161	88	1.08%
ShellsMP	26->27	305	40	13.11%	1747	480	27.48%	7408	187	2.52%
Shopping_List	26->27	236	142	60.17%	1245	323	25.94%	5715	27	0.47%
Shorty	26->27	420	261	62.14%	2898	769	26.54%	17 109	449	2.62%
Shortyz	30->31	493	143	29.01%	2958	721	24.37%	19 744	224	1.13%
Shosetsu	29->30	387	322	83.20%	2752	387	14.06%	14 784	249	1.68%
Shutdown	30->31	342	213	62.28%	1959	80	4.08%	9432	232	2.46%
Signal_Generator	30->31	108	99	91.67%	576	211	36.63%	2204	47	2.13%

(*continued on next page*)

Table A.3 (*continued*).

App name	API update	Classes			Methods			Statements		
		# Total	# Aff	% Aff	# Total	# Aff	% Aff	# Total	# Aff	% Aff
Simple_Chess_Clock	25->26	333	40	12.01%	1938	485	25.03%	7381	185	2.51%
SimpleLogin	24->25	106	79	74.53%	541	71	13.12%	2300	56	2.43%
Smart_Gadget	29->30	196	118	60.20%	1421	123	8.66%	9564	220	2.30%
SNotepad	25->26	578	105	18.17%	2470	312	12.63%	9478	44	0.46%
SparkleShare	24->25	497	348	70.02%	3082	949	30.79%	15 065	159	1.06%
SQRL_Login	27->28	229	181	79.04%	1123	332	29.56%	5547	111	2.00%
StageFever	30->31	135	91	67.41%	504	100	19.84%	2187	16	0.73%
StartFlagExploit	25->26	407	224	55.04%	2895	669	23.11%	12 408	190	1.53%
Stocks_Widget	27->28	141	79	56.03%	972	336	34.57%	4668	120	2.57%
Stoic_Reading	24->25	560	437	78.04%	4480	932	20.80%	15 395	216	1.40%
Sudoku	26->27	215	192	89.30%	1171	202	17.25%	4663	102	2.19%
Suntimes_Calendars	25->26	416	375	90.14%	2756	846	30.70%	13 268	341	2.57%
Survival_Manual	28->29	114	102	89.47%	545	22	4.04%	3032	78	2.57%
Swiftnotes	27->28	474	365	77.00%	3017	1113	36.89%	19 786	244	1.23%
Symphony	30->31	239	182	76.15%	1626	257	15.81%	11 830	191	1.61%
SyncMemories	25->26	370	271	73.24%	1591	324	20.36%	5776	23	0.40%
SyncOnWifi	26->27	446	415	93.05%	2453	791	32.25%	14 794	64	0.43%
SyncPlayer	30->31	525	336	64.00%	2993	197	6.58%	15 282	314	2.05%
SyncTool	29->30	227	144	63.44%	1489	293	19.68%	4801	30	0.62%
Synergy	30->31	154	21	13.64%	976	119	12.19%	3875	59	1.52%
TapUnlock	24->25	264	191	72.35%	1632	329	20.16%	7663	165	2.15%
Tempus_Romanum	26->27	526	190	36.12%	3419	278	8.13%	17 877	144	0.81%
TimeTable	26->27	266	179	67.29%	1563	451	28.85%	9784	94	0.96%
Todo_Agenda_for_Android_4_-_7_0	27->28	198	127	64.14%	1070	410	38.32%	5589	81	1.45%
TourCount	24->25	610	391	64.10%	3383	563	16.64%	17 853	441	2.47%
Trigger	27->28	362	261	72.10%	2205	270	12.24%	10 785	98	0.91%
Truly_Creative_live_wallpaper	30->31	379	57	15.04%	2085	607	29.11%	8594	155	1.80%
Tux_Rider	24->25	354	57	16.10%	2675	719	26.88%	11 051	230	2.08%
UART_Smartwatch	24->25	254	145	57.09%	1468	169	11.51%	5554	43	0.77%
USB_HID_Terminal	28->29	492	266	54.07%	2505	715	28.54%	9864	165	1.67%
Vanilla_Music	28->29	413	133	32.20%	2530	527	20.83%	8280	207	2.50%
Voice	26->27	167	112	67.07%	608	54	8.88%	2670	64	2.40%
Weather	27->28	468	389	83.12%	2600	685	26.35%	12 874	154	1.20%
Weather_notification	30->31	344	324	94.19%	2099	263	12.53%	9899	72	0.73%
WhatsApp_Web_To_Go	30->31	453	299	66.00%	2869	640	22.31%	9646	97	1.01%
Wi-Fi_Matic	27->28	311	84	27.01%	1131	61	5.39%	5373	41	0.76%
Wi-Fi_Privacy_Police	30->31	570	189	33.16%	3777	679	17.98%	11 731	58	0.49%
WiFi_ACE	27->28	441	186	42.18%	2499	584	23.37%	11 001	280	2.55%
WIFI_EAP_SIM_Conf	27->28	277	95	34.30%	1801	520	28.87%	6789	143	2.11%
WiFi_Walkie_Talkie	27->28	488	445	91.19%	2983	856	28.70%	13 586	185	1.36%
WiFi_Warning	30->31	545	327	60.00%	3089	456	14.76%	13 360	258	1.93%
Wikipedia	28->29	625	467	74.72%	4587	1253	27.32%	66 350	1741	2.62%
Wulkanowy	26->27	597	251	42.04%	3582	787	21.97%	11 025	114	1.03%
Youtube_Listing_App	27->28	137	44	32.12%	1199	336	28.02%	6519	118	1.81%
YubiTOTP	30->31	524	226	43.13%	2516	979	38.91%	10 515	214	2.04%
zbarcamdemo	29->30	283	179	63.25%	1353	522	38.58%	8024	190	2.37%

Table A.4
Detailed results for affected tests and untested affected code.

App name	Tests			Time (s)			Analysis overhead (s)	Affected code	
	# Total	# Aff	% Aff	Total	Aff	Saved		# Tested	# Untested
1010!_Klooni	320	87	27.19%	9836.80	2556.71	7280.09	33.00	247.97	67.00
2_Player_Battle	838	250	29.83%	32,011.60	9731.45	22,280.15	344.01	71.00	125.00
A2DP_Volume	459	350	76.25%	13,444.11	10,969.11	2475.01	294.98	248.00	22.00
Aard_2	220	55	25.00%	6388.80	1736.16	4652.64	525.82	155.00	361.00
adbWireless	114	55	48.25%	3649.14	1591.54	2057.60	32.45	11.00	7.00
AELF	350	236	67.43%	11,102.00	8077.31	3024.69	266.20	96.00	22.00
AFWall+	184	20	10.87%	7216.48	785.18	6431.30	94.33	10.00	60.00
Alarmio	344	113	32.85%	11,565.28	3837.05	7728.23	150.31	32.00	48.00
ALSA_Mixer_WebUI	199	28	14.07%	7689.36	974.81	6714.55	74.13	17.00	80.00
AmazeFileManager	153	57	37.25%	5983.47	2506.22	3477.25	486.95	380.00	323.00
Andor_s_Trail	538	81	15.06%	18,614.80	2785.78	15,829.02	234.65	59.00	267.00
AndStatus	537	357	66.48%	17,817.66	12,461.21	5356.45	217.56	157.00	39.00
AnkiDroid	415	49	11.81%	15,097.70	1823.62	13,274.08	216.50	16.00	94.00
Antimine	188	76	40.43%	5953.96	2349.15	3604.81	323.76	102.00	107.00
AnySoftKeyboard	352	180	51.14%	9926.40	5238.43	4687.97	289.80	97.00	61.00
ApnSwitch	509	231	45.38%	18,110.22	8013.51	10,096.71	141.24	145.00	120.00
ArchWiki_Viewer	124	70	56.45%	4840.96	2612.56	2228.40	53.31	15.00	6.00
Audinaut	386	102	26.42%	10,155.66	2919.78	7235.88	164.40	61.00	130.00
Auto_Airplane_Mode	235	88	37.45%	5914.95	2144.08	3770.87	175.04	17.00	20.00

(*continued on next page*)

Table A.4 (*continued*).

App name	Tests			Time (s)			Analysis overhead (s)	Affected code	
	# Total	# Aff	% Aff	Total	Aff	Saved		# Tested	# Untested
Barnacle_Wifi_Tether	151	73	48.34%	5733.47	2949.21	2784.26	168.53	135.00	96.00
Beam_File	283	100	35.34%	7145.75	2401.28	4744.48	203.38	30.00	40.00
Better_schedule	144	111	77.08%	3630.24	2608.02	1022.22	138.54	82.00	6.00
Better_Wifi_on_off	463	79	17.06%	17,765.31	3113.07	14,652.24	302.08	51.00	195.00
Blitzortung_Lightning_Monitor	456	156	34.21%	15,932.64	5052.74	10,879.90	373.05	37.00	51.00
BlitzType_Keyboard	600	396	66.00%	20,832.00	14,285.34	6546.66	324.26	50.00	12.00
Bluetooth_terminal	541	467	86.32%	15,093.90	13,381.09	1712.81	261.04	160.00	10.00
Booky_McBookface	365	244	66.85%	9420.65	5737.15	3683.50	385.14	277.00	68.00
Botifier	248	104	41.94%	8325.36	3142.15	5183.21	365.31	160.00	156.00
Camera	226	31	13.72%	6515.58	971.48	5544.10	171.15	31.00	153.00
Camp_2019	140	27	19.29%	5360.60	990.41	4370.19	58.28	13.00	42.00
CaptivePortalLogin	149	116	77.85%	4210.74	3192.93	1017.81	86.72	56.00	3.00
Car_Bus_Interface	275	154	56.00%	10,348.25	6258.62	4089.63	118.60	121.00	58.00
CineLog	145	6	4.14%	2571.50	144.70	2426.80	202.28	18.00	13.00
Clementine_Remote	336	285	84.82%	10,164.00	8302.26	1861.74	167.54	228.00	5.00
Clock_Live_Wallpaper	480	61	12.71%	15,033.60	1929.63	13,103.97	164.01	24.00	128.00
codename_hippopotamos	275	240	87.27%	6957.50	6387.74	569.76	226.25	144.00	0.00
Color_Namer	604	39	6.46%	18,428.04	1141.10	17,286.94	183.86	10.00	108.00
Comfort_Reader	137	28	20.44%	3889.43	763.92	3125.51	54.12	22.00	65.00
Congress_Fahrplan	196	56	28.57%	6675.76	1956.95	4718.81	158.37	52.00	98.00
Cool_Reader	541	259	47.87%	19,778.96	8910.37	10,868.59	159.70	84.00	61.00
Countdown_for_DashClock	559	401	71.74%	15,735.85	12,247.64	3488.21	290.87	244.00	39.00
Cover_Fetcher	561	276	49.20%	19,578.90	9227.84	10,351.06	364.74	113.00	77.00
Crossword	141	22	15.60%	4261.02	674.15	3586.87	68.65	11.00	47.00
Cuberite	414	106	25.60%	15,098.58	3950.87	11,147.71	498.45	40.00	89.00
DashClock__Sunrise	517	241	46.62%	17,536.64	7864.08	9672.56	138.92	53.00	41.00
Dialer_for_Pebble	501	154	30.74%	18,331.59	5848.98	12,482.61	206.60	128.00	218.00
Dicer	288	231	80.21%	9878.40	8636.40	1242.00	218.95	39.00	1.00
DNSSetter	137	110	80.29%	3827.78	3239.36	588.42	45.65	71.00	2.00
DroidFish	449	362	80.62%	14,736.18	12,973.88	1762.30	185.63	146.00	4.00
Duorem	127	58	45.67%	4069.08	1923.36	2145.72	73.50	34.00	28.00
DuOTP	188	119	63.30%	5136.16	3257.58	1878.58	79.71	78.00	24.00
Easy_xkcd	374	74	19.79%	10,640.30	1930.56	8709.74	116.48	19.00	58.00
EasyRSS	393	213	54.20%	11,691.75	6704.28	4987.47	158.63	84.00	45.00
Elementary	150	26	17.33%	5452.50	946.99	4505.51	17.77	4.00	11.00
Enchanted_Fortress	469	154	32.84%	12,859.98	3935.54	8924.44	443.64	36.00	55.00
Equate	124	52	41.94%	4010.16	1715.31	2294.85	39.20	15.00	13.00
Equipe_RSS	778	58	7.46%	29,929.66	2157.63	27,772.03	813.16	21.00	208.00
Eve-control	217	95	43.78%	6976.55	2779.37	4197.18	117.47	9.00	8.00
Eve-control	184	110	59.78%	6657.12	3685.29	2971.83	48.54	56.00	22.00
Exodus	112	76	67.86%	4222.40	3151.72	1070.68	63.55	41.00	9.00
EZ_Wifi_Notification	344	206	59.88%	10,161.76	6614.66	3547.10	82.87	41.00	15.00
Fabularium	308	96	31.17%	8251.32	2499.83	5751.49	223.52	25.00	40.00
Fast_Brightness_Control_Widget	875	436	49.83%	27,606.25	14,361.06	13,245.19	255.56	152.00	101.00
Fate_Sheets	839	127	15.14%	32,913.97	4977.23	27,936.74	316.13	22.00	94.00
Fire_Chat	364	136	37.36%	13,544.44	5303.47	8240.97	154.89	20.00	23.00
Flight_Mode	140	101	72.14%	4467.40	3529.09	938.31	218.14	136.00	21.00
FOSDEM_Companion	482	318	65.98%	16,214.48	11,382.16	4832.32	494.13	278.00	72.00
FOSS_Browser	333	177	53.15%	11,108.88	5916.53	5192.35	95.83	69.00	38.00
Freebloks	282	167	59.22%	7540.68	4693.32	2847.36	160.83	67.00	26.00
Giggity	284	224	78.87%	10,743.72	8177.33	2566.39	68.00	53.00	2.00
Gilga	222	57	25.68%	7920.96	1972.75	5948.21	44.68	21.00	45.00
GM_Dice	484	164	33.88%	14,011.80	4980.44	9031.36	294.38	75.00	107.00
Gnucash	141	117	82.98%	4179.62	3635.07	544.55	461.68	319.00	264.00
Great_Freedom	383	83	21.67%	9617.13	1961.17	7655.96	463.01	38.00	106.00
Hacker_s_Keyboard	219	26	11.87%	7689.09	955.76	6733.33	214.06	9.00	52.00
Hall_Monitor	485	325	67.01%	14,234.75	9996.61	4238.14	136.40	106.00	25.00
Hash_Droid	167	12	7.19%	5003.32	345.86	4657.46	296.02	5.00	51.00
Heb12	266	45	16.92%	10,469.76	1806.62	8663.14	231.46	13.00	49.00
I2P	129	44	34.11%	4473.72	1670.88	2802.84	203.88	35.00	49.00
iBeaconDetector	440	305	69.32%	11,140.80	7382.81	3757.99	185.70	229.00	46.00
Inetify	350	298	85.14%	9047.50	7757.22	1290.28	84.21	92.00	7.00
INSTEAD	177	126	71.19%	6678.21	4506.77	2171.44	272.76	168.00	28.00
J2ME_Loader	300	223	74.33%	11,886.00	9064.98	2821.02	150.62	240.00	29.00
Kakugo	228	109	47.81%	7264.08	3163.67	4100.41	217.41	43.00	31.00
KeePass_NFC	255	70	27.45%	6887.55	1807.51	5080.04	118.34	42.00	84.00
KeePassDroid	421	171	40.62%	14,739.21	5609.55	9129.66	255.91	98.00	102.00
Kiwix	692	150	21.68%	23,521.08	5098.50	18,422.58	832.57	180.00	511.00
Knock_on_Ports	109	10	9.17%	4071.15	385.83	3685.32	105.64	8.00	59.00
KouChat	205	75	36.59%	6418.55	2226.14	4192.41	329.40	76.00	96.00
Kouchat	550	133	24.18%	4874.37	1277.79	3596.59	503.04	237.00	8.00
LeMonde	265	154	58.11%	6847.60	3732.64	3114.96	244.07	59.00	25.00

(*continued on next page*)

Table A.4 (*continued*).

App name	Tests			Time (s)			Analysis overhead (s)	Affected code	
	# Total	# Aff	% Aff	Total	Aff	Saved		# Tested	# Untested
Lexica	531	304	57.25%	17,746.02	10,840.38	6905.64	349.55	163.00	73.00
Logcat_to_UDP	142	25	17.61%	3709.04	668.02	3041.02	162.56	41.00	153.00
Look4Sat	548	337	61.50%	17,108.56	11,299.70	5808.86	169.00	30.00	10.00
Mach3Pendant	303	76	25.08%	10,147.47	2522.33	7625.14	347.03	92.00	213.00
Materialistic	312	277	88.78%	9748.54	9022.41	726.13	614.74	396.00	12.00
Mensa-Guthaben	331	274	82.78%	8606.00	7686.80	919.20	246.24	180.00	1.00
microMathematics_Plus	117	68	58.12%	4238.91	2320.75	1918.16	136.68	42.00	17.00
Mines3D	750	624	83.20%	28,552.50	25,608.62	2943.88	217.20	296.00	0.00
Minetest	897	55	6.13%	28,982.07	1787.71	27,194.36	332.75	21.00	262.00
MoClock	557	369	66.25%	17,723.74	12,399.11	5324.63	142.75	39.00	10.00
Moonlight	360	131	36.39%	9990.00	3958.79	6031.21	292.32	114.00	147.00
Morse	143	35	24.48%	5628.48	1322.50	4305.98	96.13	10.00	22.00
Moscow_Wi-Fi_autologin	173	64	36.99%	4994.51	1672.15	3322.36	207.36	38.00	47.00
MTG_Familiar	650	183	28.15%	22,854.00	6369.94	16,484.06	424.22	129.00	250.00
Music	481	48	9.98%	12,073.10	1101.19	10,971.91	401.35	29.00	209.00
Music_Player	317	174	54.89%	10,476.85	6262.51	4214.34	128.90	89.00	45.00
Music_Player_GO	163	97	59.51%	6029.37	3875.07	2154.30	104.87	19.00	7.00
MusicBrainz	215	30	13.95%	6226.40	833.18	5393.22	488.40	41.00	198.00
Muzei_with_Studio_Ghibli	227	163	71.81%	7563.64	4980.37	2583.27	88.35	57.00	9.00
MyOwnNotes	170	110	64.71%	5846.30	3794.25	2052.05	192.19	104.00	29.00
NCBSinfo	203	63	31.03%	7825.65	2387.36	5438.29	93.93	27.00	43.00
NDEF_Tools_for_Android	225	79	35.11%	7602.75	2418.49	5184.26	107.75	18.00	24.00
NewsBlur	206	158	76.70%	5792.72	4296.34	1496.38	228.41	96.00	8.00
NFC_Key	252	214	84.92%	7207.20	5863.34	1343.86	388.58	50.00	0.00
NFC_Reader	882	94	10.66%	27,756.54	2739.27	25,017.27	354.12	43.00	293.00
NFC_Tag_maker	185	149	80.54%	4778.55	4068.04	710.51	115.76	141.00	4.00
NFCard	117	25	21.37%	3844.62	886.40	2958.22	116.99	23.00	65.00
NFCMessageBoard	461	224	48.59%	12,391.68	5527.39	6864.29	275.31	124.00	88.00
NitroShare	758	547	72.16%	21,019.34	13,939.68	7079.66	238.94	275.00	42.00
Note_Crypt_Pro	127	49	38.58%	3679.19	1311.65	2367.54	42.46	10.00	10.00
Notification_Cron	259	135	52.12%	9528.61	5105.72	4422.89	379.53	239.00	143.00
Number_Guesser	214	144	67.29%	7213.94	4825.11	2388.83	107.41	35.00	8.00
Odyssey	537	94	17.50%	13,457.22	2270.84	11,186.38	277.40	87.00	324.00
Offline_Calendar	496	438	88.31%	14,919.68	14,387.14	532.54	296.54	186.00	1.00
OgreSampleBrowser	263	43	16.35%	8526.46	1387.09	7139.37	144.50	8.00	31.00
OI_Notepad	224	68	30.36%	5837.44	1842.96	3994.48	208.89	22.00	37.00
OkcAgent	214	161	75.23%	8352.42	6899.65	1452.77	126.33	46.00	4.00
OMW_Nightly	133	84	63.16%	4420.92	2619.05	1801.87	44.54	25.00	7.00
Open_Sudoku	122	8	6.56%	3550.20	219.76	3330.44	113.65	5.00	50.00
Open_WiFi_Cleaner	392	86	21.94%	10,278.24	2394.73	7883.51	248.61	57.00	158.00
OpenDNSUpdater	395	309	78.23%	12,738.75	10,383.79	2354.96	242.86	126.00	8.00
OpenFoodFacts	128	16	12.50%	4499.21	723.28	3775.93	1027.24	34.00	625.00
OpenManga	281	175	62.28%	7168.31	4852.64	2315.67	290.16	268.00	90.00
OpenStud	245	88	35.92%	7913.50	3058.42	4855.08	131.55	44.00	58.00
Oscilloscope	152	8	5.26%	4739.36	249.69	4489.67	75.05	4.00	54.00
PactrackDroid	145	41	28.28%	5475.20	1518.74	3956.46	135.36	35.00	67.00
PassAndroid	483	346	71.64%	12,210.24	9429.14	2781.10	381.41	77.00	12.00
Password_Store	370	181	48.92%	10,041.80	5231.64	4810.16	452.41	191.00	133.00
PDF_Creator	992	255	25.71%	32,408.64	9163.94	23,244.71	311.41	145.00	325.00
Pdf_Viewer_Plus	330	205	62.12%	13,200.00	7995.00	5205.00	291.83	176.00	60.00
Persian_Calendar	132	70	53.03%	4419.36	2463.12	1956.24	64.85	34.00	19.00
Pixel_Wheels	128	64	50.00%	4952.32	2488.54	2463.78	44.41	56.00	37.00
Planes_Android	249	121	48.59%	7935.63	3817.71	4117.92	242.18	59.00	41.00
Port_Authority	248	213	85.89%	6666.24	6131.95	534.29	241.78	95.00	6.00
pOT-Droid	127	59	46.46%	4414.52	1999.57	2414.95	246.55	89.00	69.00
Prayer_Times__Islamic_Tools_	745	584	78.39%	21,545.40	16,956.84	4588.56	273.65	234.00	14.00
PReVo	401	283	70.57%	13,854.55	9014.99	4839.56	232.53	164.00	29.00
Progress_Bars	187	146	78.07%	6958.27	4932.86	2025.41	394.86	231.00	15.00
Protect_Baby_Monitor	526	271	51.52%	18,162.78	9198.55	8964.23	138.42	104.00	63.00
PySolFC	130	95	73.08%	3697.20	2812.57	884.63	150.38	31.00	4.00
QR_Scanner	787	241	30.62%	21,980.91	7397.51	14,583.40	265.54	160.00	274.00
QRStream	211	144	68.25%	6300.46	4566.43	1734.03	529.51	345.00	76.00
QuickDic	246	202	82.11%	7965.48	6266.05	1699.43	181.74	174.00	2.00
RadioBeacon	313	137	43.77%	8513.60	3722.67	4790.93	398.87	123.00	111.00
RadioDroid	424	186	43.87%	16,828.56	7729.31	9099.25	342.23	122.00	108.00
Reader_for_Selfoss	135	76	56.30%	3890.70	2330.50	1560.20	111.23	21.00	9.00
RedReader	319	160	50.16%	11,321.31	6092.92	5228.39	296.55	31.00	19.00
Remote_Droid	269	226	84.01%	8064.62	6450.26	1614.36	156.35	133.00	8.00
RingyDingyDingy	489	347	70.96%	15,765.36	11,925.64	3839.72	213.08	229.00	39.00
ScreenStream	218	41	18.81%	7643.08	1404.40	6238.68	150.26	17.00	55.00
SDB_Viewer	911	331	36.33%	27,466.65	9001.64	18,465.01	391.87	112.00	143.00
SealNote	491	305	62.12%	15,687.45	9579.09	6108.36	158.34	138.00	47.00

(*continued on next page*)

Table A.4 (*continued*).

App name	Tests			Time (s)			Analysis overhead (s)	Affected code	
	# Total	# Aff	% Aff	Total	Aff	Saved		# Tested	# Untested
Seeks	411	337	82.00%	15,412.50	11,474.85	3937.65	148.40	272.00	4.00
Sensors_Sandbox	488	404	82.79%	12,775.84	9878.66	2897.18	406.09	299.00	1.00
Shattered_Pixel_Dungeon	441	260	58.96%	12,087.81	7767.99	4319.82	146.90	63.00	25.00
ShellsMP	193	117	60.62%	5888.43	3619.65	2268.78	103.71	137.00	50.00
Shopping_List	262	183	69.85%	8140.34	5168.40	2971.94	171.45	23.00	4.00
Shorty	696	395	56.75%	19,752.48	11,647.29	8105.19	376.40	306.00	143.00
Shortyz	415	172	41.45%	11,009.95	4266.55	6743.40	592.32	112.00	112.00
Shosetsu	413	46	11.14%	13,050.80	1439.06	11,611.74	266.11	34.00	215.00
Shutdown	144	105	72.92%	5682.24	4205.45	1476.79	235.80	203.00	29.00
Signal_Generator	149	101	67.79%	4294.18	2776.92	1517.26	30.86	39.00	8.00
Simple_Chess_Clock	427	111	26.00%	13,471.85	3544.07	9927.78	177.14	58.00	127.00
SimpleLogin	130	112	86.15%	4654.00	3656.76	997.24	73.60	56.00	0.00
Smart_Gadget	145	46	31.72%	4016.50	1371.04	2645.46	124.33	84.00	136.00
SNotepad	593	356	60.03%	19,675.74	11,445.91	8229.83	265.38	32.00	12.00
SparkleShare	679	156	22.97%	21,551.46	4941.54	16,609.92	180.78	44.00	115.00
SQRL_Login	248	93	37.50%	7896.32	3020.34	4875.98	144.22	50.00	61.00
StageFever	106	57	53.77%	3570.08	1910.16	1659.92	37.18	11.00	5.00
StartFlagExploit	290	48	16.55%	9268.40	1549.42	7718.98	297.79	38.00	152.00
Stocks_Widget	166	100	60.24%	4838.90	2684.72	2154.19	79.36	87.00	33.00
Stoic_Reading	359	312	86.91%	9980.20	7832.26	2147.94	169.35	206.00	10.00
Sudoku	258	37	14.34%	7559.40	1140.47	6418.93	69.95	18.00	84.00
Suntimes_Calendars	552	123	22.28%	21,748.80	4380.96	17,367.84	238.82	92.00	249.00
Survival_Manual	137	26	18.98%	3874.36	706.60	3167.76	63.67	18.00	60.00
Swiftnotes	513	82	15.98%	19,299.06	2915.17	16,383.89	217.65	47.00	197.00
Symphony	342	207	60.53%	13,526.10	7769.32	5756.78	224.77	139.00	52.00
SyncMemories	223	137	61.43%	6404.56	4111.70	2292.86	173.28	17.00	6.00
SyncOnWifi	344	183	53.20%	13,508.88	7035.50	6473.38	340.26	41.00	23.00
SyncPlayer	360	318	88.33%	10,774.80	9184.62	1590.18	183.38	313.00	1.00
SyncTool	136	29	21.32%	4542.40	934.70	3607.70	115.22	8.00	22.00
Synergy	147	49	33.33%	4677.54	1481.22	3196.32	46.50	24.00	35.00
TapUnlock	229	77	33.62%	9029.47	3215.24	5814.23	160.92	67.00	98.00
Tempus_Romanum	377	280	74.27%	13,734.11	9251.76	4482.35	482.68	129.00	15.00
TimeTable	146	77	52.74%	4933.34	2341.65	2591.69	273.95	60.00	34.00
Todo_Agenda_for_Android_4_-_7_0	236	106	44.92%	8210.44	3720.93	4489.51	145.31	44.00	37.00
TourCount	812	311	38.30%	31,611.16	11,937.73	19,673.43	464.18	203.00	238.00
Trigger	441	85	19.27%	14,504.49	2941.02	11,563.47	334.34	23.00	75.00
Truly_Creative_live_wallpaper	438	248	56.62%	17,305.38	9024.40	8280.98	94.53	106.00	49.00
Tux_Rider	348	301	86.49%	11,546.64	10,286.80	1259.84	276.28	220.00	10.00
UART_Smartwatch	309	56	18.12%	9928.17	1964.81	7963.36	127.74	10.00	33.00
USB_HID_Terminal	552	476	86.23%	16,019.04	14,435.13	1583.91	128.23	161.00	4.00
Vanilla_Music	329	106	32.22%	12,047.98	4021.46	8026.52	173.88	81.00	126.00
Voice	135	91	67.41%	3408.75	2155.29	1253.46	69.42	52.00	12.00
Weather	182	56	30.77%	6022.38	1906.78	4115.60	386.22	57.00	97.00
Weather_notification	504	62	12.30%	18,144.00	2051.21	16,092.79	287.07	11.00	61.00
WhatsApp_Web_To_Go	460	132	28.70%	15,281.20	3990.39	11,290.81	221.86	34.00	63.00
Wi-Fi_Matic	129	54	41.86%	3686.82	1388.99	2297.83	155.82	21.00	20.00
Wi-Fi_Privacy_Police	416	354	85.10%	14,460.16	12,698.80	1761.36	152.50	56.00	2.00
WiFi_ACE	375	81	21.60%	9603.75	2124.20	7479.55	286.03	73.00	207.00
WIFI_EAP_SIM_Conf	127	22	17.32%	4579.62	783.01	3796.61	169.73	30.00	113.00
WiFi_Walkie_Talkie	508	125	24.61%	13,776.96	3325.59	10,451.37	298.89	55.00	130.00
WiFi_Warning	742	91	12.26%	24,233.72	2936.40	21,297.32	387.44	38.00	220.00
Wikipedia	396	294	74.24%	14,657.57	11,489.75	3167.82	1481.86	1312.00	429.00
Wulkanowy	215	97	45.12%	7297.10	3460.08	3837.02	165.38	62.00	52.00
Youtube_Listing_App	240	190	79.17%	9088.80	7720.56	1368.24	182.53	113.00	5.00
YubiTOTP	328	271	82.62%	11,020.80	9515.35	1505.45	210.30	213.00	1.00
zbarcamdemo	190	17	8.95%	7594.30	712.79	6881.51	216.65	21.00	169.00

References

Aggarwal, K., Hindle, A., Stroulia, E., 2015. GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption. In: ICSME. pp. 311–320.

Ami, A.S., Hasan, M.M., Rahman, M.R., Sakib, K., 2018. Mobicomonkey: Context testing of android apps. In: MOBILESoft. pp. 76–79.

Android Lint, http://tools.android.com/tips/lint.

Android Version History, https://en.wikipedia.org/wiki/Android_version_history.

Apicia, https://github.com/TSUMahmud/APICIA.

Arnold, R.S., Bohner, S., 1996. Software Change Impact Analysis. IEEE Computer Society Press.

Businge, J., Serebrenik, A., van den Brand, M., 2012. Survival of eclipse third-party plug-ins. In: ICSM. pp. 368–377.

Businge, J., Serebrenik, A., van den Brand, M., 2013. Analyzing the eclipse API usage: Putting the developer in the loop. In: CSMR. pp. 37–46.

Businge, J., Serebrenik, A., Van Den Brand, M.G., 2015. Eclipse API usage: the good and the bad. Softw. Qual. J. 23 (1), 107–141.

Coppola, R., Morisio, M., Torchiano, M., 2017. Scripted GUI testing of android apps: A study on diffusion, evolution and fragility. In: PROMISE. pp. 22–32.

Do, Q.C.D., Yang, G., Che, M., Hui, D., Ridgeway, J., 2016a. Redroid: A regression test selection approach for android applications. In: SEKE. pp. 486–491.

Do, Q., Yang, G., Che, M., Hui, D., Ridgeway, J., 2016b. Regression test selection for android applications. In: MOBILESoft. pp. 27–28.

Egyed, A., 2003. A scenario-driven approach to trace dependency analysis. IEEE Trans. Softw. Eng. 29 (2), 116–132.

Elbaum, S., Malishevsky, A., Rothermel, G., 2001. Incorporating varying test costs and fault severities into test case prioritization. In: ICSE. pp. 329–338.

F-Droid, https://f-droid.org.

Fazzini, M., Xin, Q., Orso, A., 2019. Automated API-usage update for android apps. In: ISSTA. pp. 204–215.

Forward compatibility, https://developer.android.com/guide/topics/manifest/uses-sdk-element.

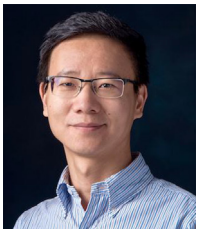
Gligoric, M., Eloussi, L., Marinov, D., 2015. Practical regression test selection with dynamic file dependencies. In: ISSTA. pp. 211–222.

Google Play's target API level requirement. https://developer.android.com/distribute/best-practices/develop/target-sdk.

Google Play Store, https://play.google.com.
Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A., 2001. Regression test selection for java software. In: OOPSLA. pp. 312–326.
Haryono, S.A., Thung, F., Kang, H.J., Serrano, L., Muller, G., Lawall, J., Lo, D., Jiang, L., 2020. Automatic android deprecated-API usage update by learning from single updated example. In: ICPC. pp. 401–405.
Haryono, S.A., Thung, F., Lo, D., Jiang, L., Lawall, J., Kang, H.J., Serrano, L., Muller, G., 2021. AndroEvolve: Automated update for android deprecated-API usages. In: ICSE-Companion. IEEE, pp. 1–4.
Hou, D., Yao, X., 2011. Exploring the intent behind API evolution: A case study. In: WCRE. pp. 131–140.
Huang, H., Wei, L., Liu, Y., Cheung, S.-C., 2018. Understanding and detecting callback compatibility issues for android applications. In: ASE. pp. 532–542.
Hyunsook Do, Rothermel, G., 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. TSE 32 (9), 733–752.
JaCoCo Java Code Coverage Library. https://www.jacoco.org/jacoco/.
Lam, P., Bodden, E., Lhoták, O., Hendren, L., 2011. The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop, Vol. 15. p. 35.
Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., Marinov, D., 2016. An extensive study of static regression test selection in modern software evolution. In: FSE. pp. 583–594.
Li, L., Bissyandé, T.F., Wang, H., Klein, J., 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In: ISSTA. pp. 153–163.
Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L., Li, X., 2017. ATOM: automatic maintenance of GUI test scripts for evolving mobile applications. In: ICST. pp. 161–171.
Li, B., Sun, X., Leung, H., Zhang, S., 2013. A survey of code-based change impact analysis techniques. Softw. Test. Verif. Reliab. 23 (8), 613–646.
Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., Poshyvanyk, D., 2013. API change and fault proneness: a threat to the success of android apps. In: ESEC/FSE. pp. 477–487.
Machiry, A., Tahiliani, R., Naik, M., 2013. Dynodroid: An input generation system for android apps. In: ESEC/FSE. pp. 224–234.
Mahmood, R., Mirzaei, N., Malek, S., 2014. EvoDroid: Segmented evolutionary testing of android apps. In: FSE. pp. 599–609.
Mahmud, T., Che, M., Yang, G., 2021a. Android compatibility issue detection using API differences. In: SANER. pp. 480–490.
Mahmud, T., Che, M., Yang, G., 2022. Android API field evolution and its induced compatibility issues. In: ESEM. Association for Computing Machinery, pp. 34–44.
Mahmud, T., Che, M., Yang, G., 2022b. ACID: An API Compatibility Issue Detector for Android Apps. In: ICSE-Companion. ISBN: 9781450392235, pp. 1–5.
Mahmud, T., Khan, M., Rouijel, J., Che, M., Yang, G., 2021. Api change impact analysis for android apps. In: COMPSAC. IEEE, pp. 894–903.
McDonnell, T., Ray, B., Kim, M., 2013. An empirical study of api stability and adoption in the android ecosystem. In: ICSM. pp. 70–79.
Migrating Apps to Android 9. https://developer.android.com/about/versions/pie/android-9.0-migration.
Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide.
Ren, X., Ryder, B.G., Stoerzer, M., Tip, F., 2005. Chianti: a change impact analysis tool for Java programs. In: ICSE. pp. 664–665.
Rothermel, G., Harrold, M.J., 1997. A safe, efficient regression test selection technique. TOSEM 6 (2), 173–210.
Ryder, B.G., Tip, F., 2001. Change impact analysis for object-oriented programs. In: PASTE. pp. 46–53.
Statista, Number of available applications in the Google Play Store from December 2009 to September 2018. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.
Test your app's compatibility with Android 11. https://developer.android.com/about/versions/11/test-changes.
Wang, J., Li, L., Liu, K., Cai, H., 2020. Exploring how deprecated python library APIs are (not) handled. In: ESEC/FSE. pp. 233–244.
Wei, L., Liu, Y., Cheung, S., 2016. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In: ASE. pp. 226–237.
Wei, L., Liu, Y., Cheung, S., 2019. PIVOT: Learning API-device correlations to facilitate android compatibility issue detection. In: ICSE. pp. 878–888.
Xiao-Bo, Z., Ying, J., Hai-Tao, W., 2011. Method on change impact analysis for object-oriented program. In: ICINIS. pp. 161–164.
Yang, G., Jones, J., Moninger, A., Che, M., 2018. How do android operating system updates impact apps? In: MOBILESoft. pp. 156–160.
Yoo, S., Harman, M., Tonella, P., Susi, A., 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: ISSTA. pp. 201–212.
Zhang, L., 2018. Hybrid regression test selection. In: ICSE. pp. 199–209.
Zhang, L., Kim, M., Khurshid, S., 2012. FaultTracer: A change impact and regression fault analysis tool for evolving Java programs. In: FSE. pp. 40:1–40:4.

Tarek Mahmud is a Ph.D. student in the Department of Computer Science at Texas State University, USA. He is working in the Software Engineering lab there. He completed his Bachelor of Science in Software Engineering (BSSE) from the Institute of Information Technology, University of Dhaka. His area of interest is Software Engineering, especially in mobile software engineering, program analysis, and software quality.

Meiru Che is a research scientist at Data61, Commonwealth Scientific and Industrial Research Organization (CSIRO), Australia. Her research interests lie in software engineering, mobile software systems, and artificial intelligence. She received her Ph.D. in Electrical and Computer Engineering from the University of Texas at Austin, USA, in 2014. She was an Assistant Professor from 2017 to 2019 and then Adjunct Professor from 2019 to 2021 in the Department of Computer Science at Concordia University Texas. She was a Lecturer in the Department of Electrical and Computer Engineering at the University of Texas at Austin from 2015 to 2017.



Guowei Yang is a Senior Lecturer in the School of Information Technology and Electrical Engineering at The University of Queensland, Australia. His research interests are in software engineering, cyber security, programming languages, and formal methods, with a focus on improving software reliability and security. He holds a Ph.D. degree in Electrical and Computer Engineering from the University of Texas at Austin, USA. In 2013, he started to work as an Assistant Professor at the Department of Computer Science, Texas State University, USA, and was promoted to be a tenured Associate Professor in 2019. He has published more than 40 papers in international conferences and journals, and served on program committees or organizing committees for top ranked conferences in software engineering, including ICSE, ESEC/FSE, ASE, and ISSTA.