William & Mary
CHARTERED 1693

W&M ScholarWorks

Arts & Sciences Articles

Arts and Sciences

2015

The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps

Gabriele Bavota

Mario Linares-Vasquez
College of William and Mary

Carlos Eduardo Bernal-Cardenas
College of William and Mary

Denys Poshyvanyk
College of William and Mary

Massimiliano Di Penta

See next page for additional authors

Follow this and additional works at: https://scholarworks.wm.edu/aspubs

Recommended Citation

Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C. E., Di Penta, M., Oliveto, R., & Poshyvanyk, D. (2014). The impact of api change-and fault-proneness on the user ratings of android apps. IEEE Transactions on Software Engineering, 41(4), 384-407.

This Article is brought to you for free and open access by the Arts and Sciences at W&M ScholarWorks. It has been accepted for inclusion in Arts & Sciences Articles by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Authors

Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Denys Poshyvanyk, Massimiliano Di Penta, and Rocco Oliveto

This article is available at W&M ScholarWorks: https://scholarworks.wm.edu/aspubs/993

The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps

Gabriele Bavota, Mario Linares-Vásquez, *Member, IEEE*, Carlos Eduardo Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk, *Member, IEEE*

Abstract—The mobile apps market is one of the fastest growing areas in the information technology. In digging their market share, developers must pay attention to building robust and reliable apps. In fact, users easily get frustrated by repeated failures, crashes, and other bugs; hence, they abandon some apps in favor of their competition. In this paper we investigate how the fault- and change-proneness of APIs used by Android apps relates to their success estimated as the average rating provided by the users to those apps. First, in a study conducted on 5,848 (free) apps, we analyzed how the ratings that an app had received correlated with the fault- and change-proneness of the APIs such app relied upon. After that, we surveyed 45 professional Android developers to assess (i) to what extent developers experienced problems when using APIs, and (ii) how much they felt these problems could be the cause for unfavorable user ratings. The results of our studies indicate that apps having high user ratings use APIs that are less fault- and change-prone than the APIs used by low rated apps. Also, most of the interviewed Android developers observed, in their development experience, a direct relationship between problems experienced with the adopted APIs and the users' ratings that their apps received.

Index Terms—Mining software repositories, empirical studies, android, API changes

1 Introduction

According to a recent study by VisionMobile [1], the mobile handset industry has been growing at 23 percent Compound Annual Growth Rate (CAGR)[1] in revenues since 2009, and the expected growth from 2012 to 2016 will be 28 percent CAGR [3]. The "App" economy is a tremendous success: iOS, BlackBerry, and Android were the most lucrative software platforms in 2012, with average monthly revenue of over $4,800, $3,700, and $3,300 per app, respectively [4]. Additionally, the developers' mindshare index during the last four years (2010-2013) shows that Android and iOS are the top two software platforms being used by developers worldwide [1], [3], [4].

What are the hidden forces that contribute to the app economy's success? Typical answers are: ubiquitous computing, low cost of handsets (especially, the Android devices), monetization models, customers' loyalty to brands such as iPhone or BlackBerry, etc. However, beyond explaining the "hidden forces" that drive consumer/developer decisions and define the reasons for the success of the apps, that success can be influenced by the software infrastructure that developers use to build apps (i.e., Application Programming Interfaces—APIs). APIs encapsulate the complexity of low-level programming details, and provide developers with a high-level model for using the underlying hardware. However, the ease-of-use of these APIs is impacted by factors related to API design and quality. For instance, top categories of API learning obstacles are related to learning resources (e.g., documentation, or code examples) and API structure (e.g., design or name of API elements) [5]. Also, APIs not ensuring backward compatibility support are typically hard to use because of their instability [6], and API breaking-changes could introduce bugs into the client code. Moreover, since developers often assume correctness behind underlying APIs, faults in APIs can drastically impact the client code quality as perceived by the end-users; on the other hand, developers avoid to use new version of APIs to skip bugs in the new version [7]. For example, Zibran et al. [8] found that among 1,513 bug reports related to various components of Eclipse, GNOME, MySQL, Python 3.1, and Android projects, 562 bug-reports were related to API usability issues; and about 175 (31 percent) of those issues were related to API correctness. Also Businge et al. [9] found that 44 percent of 512 Eclipse third-party plug-ins depends on "bad" (i.e., unstable, discouraged, and unsupported) APIs and that developers continue using those APIs. *Although one can possibly assume that API instability (change-proneness) and fault-proneness may impact the success of software applications, to the best of our knowledge such relations have not been empirically investigated yet*.

Stability and fault-proneness in the Android API is a sensitive and timely topic, given the frequent releases and the number of applications that use these APIs. Therefore, the goal of this paper is to provide solid empirical evidence and shed some light on the relationship between the success of apps (in terms of user ratings), and the

1. For a definition of CAGR see [2].

- *G. Bavota is with the Department of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy. E-mail: gabriele.bavota@unibz.it.*
- *M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk are with the Department of Computer Science, The College of William and Mary, Williamsburg, VA 23185. E-mail: {mlinarev, cebernal, denys}@cs.wm.edu.*
- *M. Di Penta is with the Department of Engineering, University of Sannio, Benevento, Italy. E-mail: dipenta@unisannio.it.*
- *R. Oliveto is with the Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy. E-mail: rocco.oliveto@unimol.it.*

Manuscript received 24 Jan. 2014; revised 20 Oct. 2014; accepted 28 Oct. 2014. Date of publication 3 Nov. 2014; date of current version 17 Apr. 2015. Recommended for acceptance by F. Tip.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2367027

0098-5589 © 2014 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

change- and fault-proneness of the underlying APIs (i.e., Android API and third-party libraries). We designed two case studies. In the first study we analyzed to what extent the APIs fault- and change-proneness affect the user ratings of the Android apps using them, while in the second we investigated to what extent Android developers experience problems when using APIs and how much they feel these problems can be causes of unfavorable user ratings/comments.

The first study (in the following referred as "Study I") was conducted on a set of 5,848 Android free apps belonging to different domains. We estimated the success of an app based on the ratings posted by users in the app store (Google Play[2]). Then, we identified the APIs used by those apps, and computed the number of bug fixes that those APIs underwent. In addition to the bug fixes, we computed different kinds of changes occurring to such APIs, including changes in the interfaces, implementation, and exception handling. Finally, we analyzed how the user ratings of an app are related to APIs fault- and change- proneness, specifically to different kinds of changes occurring to APIs. This study has mainly the aim of providing possible quantitative evidence about relationship between APIs fault- and change-proneness, and the apps' ratings. However, especially because we have no visibility over the source code of such apps and of their issue trackers, it is difficult to provide a strong rationale and, possible, a cause-effect relationship for such findings.

In order to provide explanations to the finding of Study I, we conducted a second study (in the following referred as "Study II"). This study consists of a survey, and it involved 45 professional Android developers. We asked such developers to fill-in a questionnaire composed by 21 questions organized into five categories: (i) developer's background, (ii) factors negatively impacting user ratings, (iii) frequent reasons causing bugs/crashes in Android apps, (iv) experiences with used APIs, and (v) impact of problematic APIs on the user ratings. Then, we quantitatively analyzed the answers to 19 questions by using descriptive statistics, and completed the analysis with qualitative data gathered from the other two questions (See Table 9).

It is important to point out that this work does not claim a cause-effect relationship between APIs fault- and change-proneness and the success of apps, which can be due to several other internal (e.g., app features and usability) and/or external (e.g., availability of alternative similar apps) factors. Instead, the purpose of our study is to investigate whether the change- and fault-proneness of APIs used by the app relates (or not) to the app success, measured by its ratings. That is, a heavy usage of fault-prone APIs can lead to repeated failures or even crashes of the apps, hence encouraging users to give low ratings and possibly even abandoning the apps. Similarly, the use of unstable APIs that undergo numerous changes in their interfaces can cause backward compatibility problems or require frequent updates to the apps using those APIs. Such updates, in turn, can introduce defects into the applications using unstable APIs.

Results of our first study demonstrate that Android apps having higher user ratings generally use APIs that are less fault- and change-prone than APIs used by low rated apps. For instance, *among the 5,848 analyzed apps, the 50 least successful apps use APIs that are 457 percent more fault-prone and 315 percent more change-prone, on average, than APIs used by the 50 most successful apps*. Moreover, results of our survey conducted with Android developers indicate that *62 percent of them observed, in their development experience, a direct relationship between problems experienced with the used APIs and bad users' ratings/comments*.

Structure of the paper. Section 2 defines Study I and its research questions, while Section 2 reports and discusses the results achieved from a quantitative and qualitative point of view. Section 3 presents the design and the results achieved in Study II (i.e., the survey). Section 4 discusses the threats that could affect the validity of the results achieved in both studies. Section 5 relates this work to the existing literature, while Section 6 concludes the paper and outlines directions for future work.

2 Study I: Mining Software Repositories

The *goal* of this study is to understand to what extent the APIs fault- and change-proneness affect the user ratings of the Android apps using them. The *context* consists of 5,848 free apps from the Google Play Market, and the *quality focus* is the success of those apps in terms of ratings expressed by users on the market.

2.1 Study Design

In the following we describe in detail the design and planning of the study, and in particular the context selection, the research questions, the independent and dependent variables, the data extraction process, and the analysis method.

2.1.1 Context Selection

Table 1 reports characteristics of the 5,848 apps that we analyzed. As it can be seen from the table, the apps belong to a pretty varied (30) set of categories. For each category considered in our study (e.g., photography, medical, games, etc), the table lists (i) the number of apps analyzed from the category (column #apps), (ii) the size range of the analyzed apps in terms of number of classes (column #classes), and bytecode size in terms of thousands of lines of code (KLOC). There are multiple factors that lead us to the selection of the set of apps mentioned above. First and foremost, we deliberately restricted our attention to free apps for practical reasons (paid apps would clearly require a fee). To collect free apps, we built a Crawler downloading free Android apps. We ran the Crawler for one week and collected 25,869 apps. We only considered apps having at least 10 votes to prune out unreliable ratings. With a smaller number of ratings, there was a higher risk that our results may depend on the subjectiveness of the ratings themselves. That is, if an app receives only one or two votes, the fact that they are extremely positive or negative can depend too much on the subjective reasons of those particular users. This filtering process led to the 7,097 apps considered in our previous paper [10]. Also, we excluded all the apps for which we were not able to convert their Android PacKage (APK) file into a JAR (more details can be found in Section 2.1.4). In particular, 300 apps were discarded due to errors

2. http://play.google.com verified on January 2014.

TABLE 1
Characteristics of the Apps (Grouped by Category) Used in Our Study

Category	#apps	Classes	KLOC
Arcade	265	7-566	115-6 K
Books and reference	139	7-78	1K-11 K
Brain	313	5-572	14K-31 K
Business	139	8-226	4K-16 K
Cards	189	8-633	367-4 K
Casual	313	6-566	2K-6 K
Comics	13	16-43	1K-1 K
Communication	144	6-11	117-10 K
Education	305	6-87	1K-4 K
Entertainment	603	2-11	173-20 K
Finance	158	4-107	2K-48 K
Health and fitness	41	6-104	2K-7 K
Libraries and demo	128	1-310	11K-56 K
Lifestyle	370	2-572	1K-3 K
Media and video	232	5-572	2K-8 K
Medical	5	13-107	2K-21 K
Music and audio	239	2-190	3K-53 K
News and magazines	177	5-280	805-2 K
Personalization	528	4-29	557-23 K
Photography	199	7-1974	35K-132 K
Productivity	137	7-217	4K-7 K
Racing	190	15-280	6K-48 K
Shopping	45	5-114	2K-38 K
Social	41	9-318	4K-7 K
Sports	183	7-280	5K-6 K
Sports games	167	6-572	14K-20 K
Tools	484	3-65	1K-11 K
Transportation	23	12-144	1K-3 K
Travel and local	74	8-251	5K-44 K
Weather	4	5-41	871-11 K
Total	**5,848**	**2-572**	**1K-132 K**

during the conversion from APK to JAR. Finally, we limited our attention to a subset of apps using APIs (both Android SDK APIs and third-party APIs) for which it was possible to retrieve the change history from a versioning system. This resulted in the removal of other 1,249 apps, leading to the final 5,848 apps.t

2.1.2 Research Questions

In the context of this study (i.e., Study I) we formulated the following two research questions:

- **RQ**$_1$: *Does the Fault-Proneness of APIs Affect the User Ratings of Android Apps?* This research question aims at investigating if Android apps having lower user ratings make heavier use of fault-prone APIs than apps having higher user ratings. The conjecture is that the usage of fault-prone APIs can cause annoying failures and crashes, and for this reason users provide low ratings. Specifically, we test the following null hypothesis:

  H_{0_1}: *There is no significant difference between the average fault-proneness of APIs used by apps with high and low rates.*
- **RQ**$_2$: *Does the Change-Proneness of APIs Affect the User Ratings of Android Apps?* This research question is similar to **RQ**$_1$, however it considers the change-proneness instead of the fault-proneness as the main factor to analyze. The conjecture is that if APIs change a lot, such changes may alter their behavior and even worse their interface, hence having a side effect on the applications using them. First, an evolved API may not be back-compatible with a previous version, and therefore could alter the app behavior in an undesired way. Second, changes in API signatures may require adaptations on the app's side that, in turn, could induce faults. Thus, the null hypothesis being tested is

  H_{0_2}: *There is no significant difference between the average change-proneness of APIs used by apps with high and low rates.*

2.1.3 Study Variables

The **dependent variable** for both research questions is represented by the average (mean) rating provided by the users for those apps, representing a proxy to measure the success of the considered apps. Such ratings are posted by users on the Android market as a discrete value ranging between one and five stars.

The **independent variable** considered to answer **RQ**$_1$ is the number of bugs fixed in the APIs used by the apps during the investigated time period. The analysis is restricted to the period of time going from the date in which the considered app version was released until the date in which either (i) the app has been superseded by a new version or (ii) the last rating for such app was collected, i.e., the last observation for our dependent variable.

For **RQ**$_2$ the **independent variables** are the number of changes performed in APIs used by the considered apps, measured in the same time period adopted for the fixed bugs. Specifically, we computed the following variables:

- The overall number of method changes.
- The number of changes in method signatures (method names, parameters, return types, visibility).
- The number of changes to the set of exceptions thrown by methods, as detected by analyzing their signatures. Such kind of change is particularly important to analyze because a better usage of exception handlers may improve the apps' robustness.

Note that for all changes we separately computed data for *all methods* and *public methods*. Changes to public methods were analyzed apart in our study because these methods represent the API public interface that is directly called by the apps. Similarly to **RQ**$_1$, the analysis of changes was performed in the same time period considered for bug fixes.

2.1.4 Data Extraction Process

The data needed to answer our research questions are (i) the user ratings of the 5,848 considered apps, (ii) the list of APIs used by each app, and (iii) the bug and change history of those APIs. The user ratings were downloaded from Google Play by selecting ratings related to each app version considered in our study. We mined the users' reviews just the day before we started the data analysis, in order to gather as many ratings as possible for each app considered in our study. However, in the period of time going from the date when we downloaded the apps' APK (D_1), until the date we collected the apps' ratings

(D_2), new versions of the considered apps may have been released. Thus, there was the risk of including in our analysis reviews that were not related to the specific version of the apps considered in our study. For this reason, in the period of time going from D_1 to D_2 we mined the Google Play market at time intervals of one week to verify if new versions of the considered apps were issued. As explained before, we just considered reviews in the period of time going from the date in which the considered app version was released until the date in which either the app was superseded by a new version or the last rating for such app was collected (i.e., D_2).

To identify APIs used by the apps in our study, we downloaded their Android PacKage files using a third party library.[3] An APK file is a variant of a JAR archive containing, among other information, the compiled classes in the *dex* (Dalvik EXecutable) format used by the process virtual machine in Android.

For extracting API calls from the APK files we adopted the following process:

1) we converted the APK files to JARs using the *dex2jar*[4] disassembler tool.
2) we extracted references/calls to API classes from `.class` files, using the JClassInfo[5] tool.

Once we collected the list of APIs for each app, we mined the APIs change history from their versioning systems.[6] We analyzed 85,636 developers' commits performed in a period going from October 2007 to September 2013 for a total of 39,718 bug-fixing activities and 1,082,362 method's changes. More specifically, we mined 2,105 days of history of the Android SDK APIs and, on average, 778 days for the considered third-party APIs; the number of analyzed commits is 35,702 for the Android SDK APIs (involving a total of 1,068 developers) and 49,934 for the third party APIs (by 1,232 developers). The average size of a commit in terms of number of modified files is 15 for the Android SDK APIs and nine for the third-party APIs, while the commits' frequency in terms of number of commits per month is 164 for the Android SDK APIs and 14 for the third-party APIs. Thus, the Android SDK APIs evolve much quicker than the considered third-party APIs.

In order to identify bug-fixing commits activities we used an approach proposed by Fischer et al. [11], i.e., by mining regular expressions containing issue IDs and the keyword "fix" in the commit notes, e.g., *"fixed issue #ID"* or *"issue ID"*.

For the changes, we used a code analyzer developed in the context of the MARKOS European project[7] to compare the APIs before and after each commit at a fine-grained level. In particular, while the versioning system logs just report the changes at file level granularity performed in a commit, we used the *MARKOS code analyzer* to capture changes at method level.

The code analyzer parses source code by relying on the *srcML* toolkit [12], and categorizes changes occurring in methods into three types: (i) generic change (including all kinds of changes); (ii) changes applied to the method signature (i.e., visibility change, return type change, parameter added, parameter removed, parameter type change, method rename); and (iii) changes applied to the set of exceptions thrown by the methods. Moreover, we distinguished between changes performed to *public* methods directly used by the apps and changes performed to *non public* methods. To distinguish cases where a method was removed and a new one added from cases when a method was renamed (and possibly its source code changed), the MARKOS code analyzer uses a heuristic that maps methods with different names if their source code is similar based on a metric fingerprint similar to the one used in metric-based clone detection [13]. In particular, each method is associated to a 12 digits fingerprint containing the following information: LOCs, number of statements, number of *if* statements, number of *while* statements, number of *case* statements, number of *return* statements, number of specifiers, number of parameters, number of thrown exceptions, number of declared local variables, number of method invocations, and number of used class attributes (i.e., instance variables). The accuracy of such heuristic has been evaluated by manually checking 100 methods reported as moved by the MARKOS code analyzer. Results showed that 89 of them were actually moved methods. Typical cases of false positives were those in which a method was removed from a class and a very similar one—in terms of signature and fingerprint—was added to another class.

After having analyzed the APIs, we used such information to compute, for each app, the total number of bugs fixed in the used APIs and the number of changes along the three categories mentioned above.

It is important to note that, while in our previous work [10] we focused the attention only on the official Android APIs, here we also consider all the (open source) third-party APIs used by the apps; in fact, across the 5,848 apps object of our study, 1,224 (21 percent) make use of open source third-party APIs. Our choice of also considering third-party APIs explains why we focus our study on a smaller set of apps with respect to the work in [10] (i.e., 5,848 against 7,097—82 percent). Indeed, we only consider an app in our study if it (i) does not use any third-party library or (ii) uses third-party APIs for which we were able to find the versioning system. In other words, apps using third-party APIs for which we were not able to find the versioning system were discarded by our study. In total, we were able to analyze the entire change history of 68 projects used as third-party APIs by the Android apps in addition to the official Android APIs. The list of the analyzed third-party libraries is reported in Table 2. The total number of API classes considered in this study is 19,763 compared to the 4,816 considered in [10]. Note that commercial third-party APIs were not taken into account given the impossibility to analyze their change history.

3. http://code.google.com/p/android-market-api verified on January 2014.

4. http://code.google.com/p/dex2jar verified on January 2014.

5. http://jclassinfo.sourceforge.net verified on January 2014.

6. We mined the change history of APIs with versioning systems publicly available. The list of APIs we mined is within our online appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2014.2367027.

7. www.markosproject.eu verified on January 2014.

TABLE 2
Analyzed Third-Party Libraries

API name	#apps using it	#Classes	KLOC
ACRA	152	51	8
AdWhirl	352	75	18
AndEngine	26	596	66
android-wheel	12	25	3
AndroidAsynchronousHttpClient	10	19	4
AndroidPulltorefresh	6	36	6
AndroidQuery	10	66	20
ApacheCommonsCodec	96	107	28
ApacheCommonsIO	29	200	50
ApacheCommonsLang	25	242	114
ApacheCommonsLogging	121	65	12
ApacheCordova	8	98	15
ApacheJamesMime4j	2	270	38
asmack	4	15	3
BeInToo	1	123	22
cwac-adapter	8	2	1
cwac-anddown	11	2	1
cwac-colormixer	11	6	1
cwac-endless	9	7	1
cwac-layouts	3	4	1
cwac-loaderex	3	15	1
cwac-locpoll	5	5	1
cwac-merge	8	3	1
cwac-sacklist	11	2	1
cwac-wakeful	10	5	1
DiskLRUCache	1	5	2
Droid-Fu	34	74	11
Facebook	630	156	43
FasterXMLJackson	1	143	37
google-gson	230	184	27
GoogleGDataClient	31	1,228	214
GoogleGuava	26	1,648	392
GoogleGuice	5	510	72
GoogleProtocolBuffers	2	45	21
GoogleZXing	35	459	202
ImageViewZoom	18	22	2
JodaBeans	5	135	27
JodaMoney	2	35	14
JodaPrimitives	3	154	40
JodaTime	13	317	140
JSONsimple	70	14	2
jsoup	31	80	18
JTwitter	17	90	20
JTwitterJSON	9	91	20
libgdx	324	2,035	324
Madvertise	78	10	3
MobFox	2	54	10
MongoDBJavaDriver	21	258	45
MoPub	80	154	17
NewQuickAction	6	6	1
NewQuickAction3D	10	5	1
NineOldAndroids	1	47	11
OpenUDID	6	2	1
ormlite	15	20	3
RoboGuice	10	133	10
ScribeOAuth	46	137	7
SignPostOAuth	98	54	5
slf4j	147	221	26
SlidingMenu	1	33	4
Socialize	12	977	116
SpringFramework	7	182	33
TapIt	11	50	8
Twitter4J	118	397	56
TwitterAPIME	2	125	23
UniversalImageLoader	2	78	10
ViewPagerindicator	1	42	4
WapStartPlus1	1	19	3
XMLPullParsing	396	59	12

2.1.5 Analysis Method

To define the analysis method it is important to analyze the distribution of high and low rated apps in our dataset. Fig. 1 reports the distribution of the average ratings assigned by users to these apps. Note that the number of ratings received by each app varies between 10 (the minimum we considered) and 432,900, with a first quartile $= 31$, median $= 105$, third quartile $= 597$, and mean $= 2{,}540$.

In general, the user ratings are very high: 3,251 apps (55.59 percent) exhibit an average rating greater than 4 stars. Nevertheless, due to quite large corpus of apps considered in our study, we also have 425 apps with an average rating lower than 3 stars. Thus, we can verify a possible relationship between fault- and change-proneness of used APIs and apps average user rating. One might be tempted to believe that such apps received high scores because of being free, i.e., the user is less disappointed when an app is unreliable or useless, because she did not spend money for it or, on the contrary, a good and free functionality is highly rewarded. To verify this conjecture, we analyzed the ratings for 5,848 paid (non-free) apps randomly selected from the Google Play Market.[8] Fig. 2 depicts the distribution of ratings for these commercial apps. The number of ratings received by each commercial app vary between 10 and 96,460, with a first quartile $= 16$, median $= 30$, third quartile $= 85$, and mean $= 267$. As in the case of the free apps, user ratings are generally very high: 3,359 commercial apps (57.44 percent) exhibit an average rating greater than 4 stars. Also, similarly to free apps, 438 commercial apps have an average rating lower than 3 stars. In summary, the average rating for free apps is 3.97, whereas for paid apps it is 4.02. Although Mann-Whitney test reports a significant difference between the two distributions (p-value < 0.0001), the difference has a negligible effect size (Cliff's $d = 0.05$).

Coming back to the 5,848 free apps object of our study, we group them in three different sets on the basis of their average user rating (r_a). In particular, given $Q_1 = 3.667$ and $Q_3 = 4.395$ the first and the third quartile of the distribution of the average user ratings assigned to the 5,848 apps considered in our study, we cluster the apps into the following three sets:

1) Apps having *high rating*: apps having $r_a > Q_3$.
2) Apps having *medium rating*: apps having $Q_3 \geq r_a > Q_1$.
3) Apps having *low rating*: apps having $r_a \leq Q_1$.

To address our research questions, we use descriptive statistics to provide an overview of data, then followed by the use of statistical tests and effect size measures. First, we depict boxplots of the distribution of the average number of faults and changes for APIs used by apps that received average scores in the three categories described above. It is very important to note that, for each app, we compute the average (mean) number of changes across all APIs used by that app. In this way, we do not bias the study because of apps using too many (and possibly change-prone) or too few (and possibly stable) APIs. Then, we plot and compare distributions of such averages.

8. Further information about these apps is in our online appendix, available in the online supplemental material.

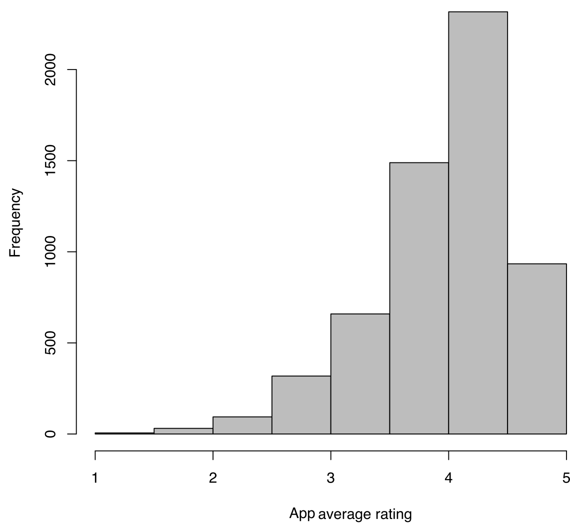


Fig. 1. Average user ratings for the 5,848 analyzed apps.

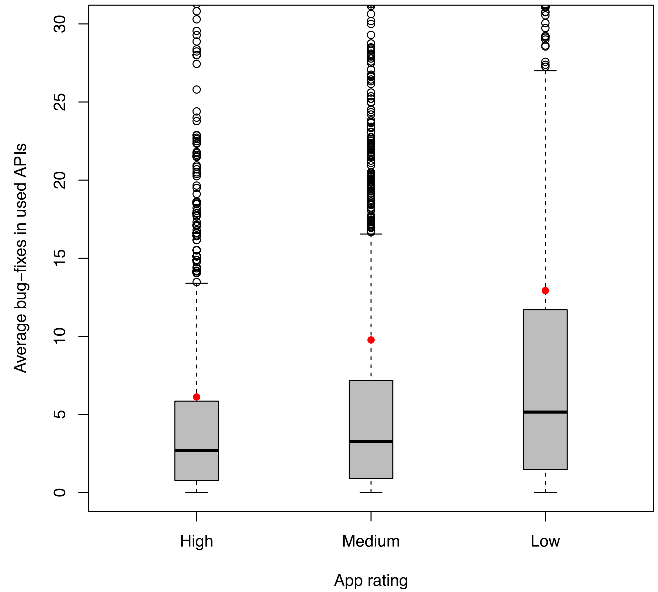


Fig. 3. Boxplots of average number of bug fixes in API classes used by apps having different levels of rating. The red dot indicates the mean.

In addition to showing boxplots, we compare such distributions using Mann-Whitney test [14]. For the latter, we pairwise compared the fault-and change-proneness for the three groups. The results were statistically significant at $\alpha = 0.05$. Since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [15]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n-1$, and so on.

We also estimated the magnitude of the difference between fault- and change-proneness of the APIs used by different groups of apps; we used the Cliff's Delta (or d), a non-parametric effect size measure [16] for ordinal data. We followed the guidelines in [16] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

2.1.6 Replication Package

The data set used in our study is publicly available at http://www.cs.wm.edu/semeru/data/tse-android/.Specifically, we provide: (i) the list (and URLs) of the studied 5,848 apps, together with the user ratings distributions; (ii) the list of APIs used by each app; (iii) complete information on the bugs fixed and changes that occurred in the APIs considered in our study (both official Android as well as third-party APIs); (iv) the *R* scripts and working data sets used to run the statistical tests and produce the plots and tables presented.

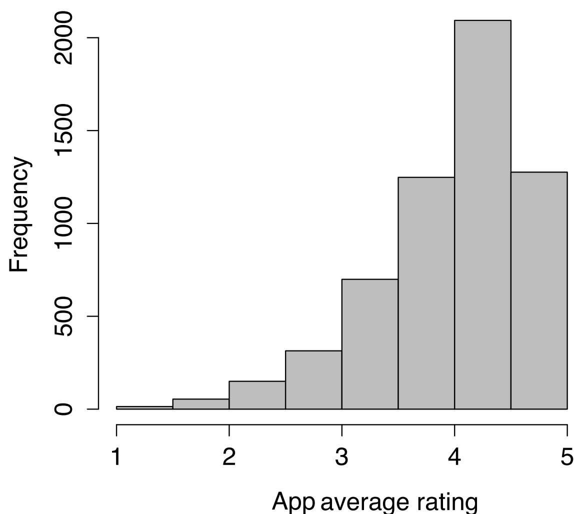


Fig. 2. Average user ratings for 5,848 paid apps.

2.2 Results

This section reports the results aimed at answering the two research questions formulated in Section 2.1.2.

2.2.1 Does the Fault-Proneness of APIs Affect the User Ratings of Android Apps?

Boxplots in Fig. 3 show the distribution of average number of bug fixes in API classes used by apps having different levels of rating (i.e., *high*, *medium*, and *low* rating as defined in Section 2.1.5). Note that we set 30 as a limit for the y-axis (i.e., average number of bug fixes in API classes) for readability purposes.

The boxplots reported in Fig. 3 highlight that apps having a higher average user rating use APIs having a lower bug-proneness. In particular, apps having a *high* rating use APIs with 6.1 bug-fixes on average. This number grows up to 9.8 (+61 percent) for apps having a *medium* rating and reaches 12 (+111 percent) for apps having a *low* rating. Overall, the difference in terms of APIs fault-proneness between apps having different levels of rating is very clear by looking to the distributions depicted in Fig. 3.

We also compared the difference in terms of API bugs between the 50 most and the 50 least successful apps (in terms of achieved average user rating). The 50 most successful apps are those having an average rating higher than 4.946, while the 50 least successful exhibit an average rating lower than 2.068. For the former, the average number of bug fixes in the used APIs is 4.4, while for the latter we

TABLE 3
Use of Fault-Prone APIs by Apps Having Different Levels of Rating: Mann-Whitney Test (adj. p-value) and Cliff's Delta (d)

Test	adj. p-value	d
highrating *vs* medium rating	<0.0001	0.10 (Small)
highrating *vs* low rating	<0.0001	0.37 (Medium)
mediumrating *vs* low rating	<0.0001	0.18 (Small)

measured an average of 24.5 bug fixes in the used APIs (+457 percent).

Table 3 reports the results of the Mann-Whitney test (p-value) and the Cliffs d effect size. We compared each set of apps (grouped by level of rating) with all other sets having a lower rating (e.g., *high rating vs*. the other). As we can see from the table, apps having a higher rating always exhibit a statistically significant lower number of bug fixes in the used APIs than apps having a lower rating (p-value always < 0.0001). The Cliff's d is small (0.10) when comparing apps having a *high* rating and apps having a *medium* rating, and medium (0.37) when the comparison is performed between apps having a *high* rating and apps having a *low* rating. The effect size is small ($d = 0.18$) when comparing apps having a *medium* rating and those having *low* rating. As expected, also the comparison of the 50 most and the 50 least successful apps shows statistical significant difference, with a p-value < 0.0001 and a large effect size ($d = 0.66$).

With the achieved results, we can reject our null hypothesis H_{0_1}, i.e., APIs used by apps having higher user ratings are, on average, significantly less fault-prone than APIs used by low rated apps. However, it is interesting to understand if the observed difference in terms of APIs faultproneness between apps having different levels of rating is due to the used official Android APIs, third-party APIs, or to both of them. To this aim, we separately investigated the fault-proneness of the official Android APIs and of the third-party APIs used by the apps object of our study.

Concerning the official Android APIs, apps having a *high* rating use APIs that underwent, on average, 6.2 bug fixes, as compared to the 9.7 (+56 percent) of apps having a *medium* rating and the 13.0 (+109 percent) of apps having a *low* rating. This result is inline with what we observed when analyzing all the used APIs as a whole. Also the results of the Mann-Whitney test reported in Table 4 confirm that official Android APIs used by apps having a higher average user rating are, on average, significantly less fault-prone than APIs used by low rated apps. Indeed, as already observed when considering all APIs, apps having a higher rating always exhibit a statistically significant lower number of bug fixes in the used APIs than apps having a lower rating (p-value always < 0.0001). In this case, the effect size is small in all comparisons.

TABLE 4
Use of Fault-Prone **Android** API by Apps Having Different Levels of Rating: Mann-Whitney Test (adj. p-value) and Cliff's Delta (d)

Test	adj. p-value	d
highrating *vs* medium rating	<0.0001	0.10 (Small)
highrating *vs* low rating	<0.0001	0.27 (Small)
mediumrating *vs* low rating	<0.0001	0.18 (Small)

TABLE 5
Use of Fault-Prone **Third-Party** APIs by Apps Having Different Levels of Rating: Mann-Whitney Test (adj. p-value) and Cliff's Delta (d)

Test	adj. p-value	d
highrating *vs* medium rating	<0.0001	0.09 (Small)
highrating *vs* low rating	<0.0001	0.27 (Small)
mediumrating *vs* low rating	<0.0001	0.19 (Small)

When analyzing third-party APIs in isolation we only considered the 1,224 apps using at least one third-party API since, as explained in Section 2.1.4, not all the considered apps use third-party APIs. In this case we observed a slightly different trend:

- apps having a *high* rating use third-party APIs subject, on average, to 1.3 bug-fixing activities.
- apps having a *medium* rating use third-party APIs subject, on average, to 3.6 bug-fixing activities (+177 percent).
- apps having a *low* rating use third-party APIs subject, on average, to 2.7 bug-fixing activities (+108 percent).

Thus, while it is confirmed that apps having a *high* rating use less fault-prone APIs than apps having a *medium* and a *low* rating, from the average values it seems that apps having a *medium* rating use APIs more fault prone than those used by apps having a *low* rating. However, by looking into the data we found that this result is mainly due to a set of 28 apps falling in the *medium* rating category and all using the same (fault-prone) third-party APIs. In particular, these 28 apps are developed by the same software house[9] and use APIs subject to a number of bug-fixes going from a minimum of 23 to a maximum of 46, clearly raising the average value of bug-fixes in the *medium* rating category. In fact, when comparing the fault-proneness of the three categories by using the Mann-Whitney test (see Table 5), we obtain that apps having higher ratings use APIs statistically significant less fault-prone than low rated apps, even when comparing apps having a *medium* rating with those having a *low* (p-value always <0.0001, with a small effect size).

Summarizing, the results of our $\mathbf{RQ}_1$ show that the higher the rating of the apps, the lower the fault-proneness of the APIs they use. This holds when considering all APIs, as well as the official Android APIs and third-party APIs in isolation.

2.2.2 Does the Change-Proneness of APIs Affect the User Ratings of Android Apps?

Boxplots in Fig. 4 show the change-proneness of APIs used by the three different sets of apps considered in our study. In particular, Figs. 4a and 4b report the overall number of method changes and the overall number of changes in the method signatures, respectively, while Figs. 4c and 4d show the same data by considering the APIs' public methods only.

Fig. 4 suggests that apps having a higher rating generally use more stable APIs, i.e., APIs having a lower

9. http://www.androidpit.it/it/android/market/applicazioni/list/owner/LightCubeMagic verified on January 2014.

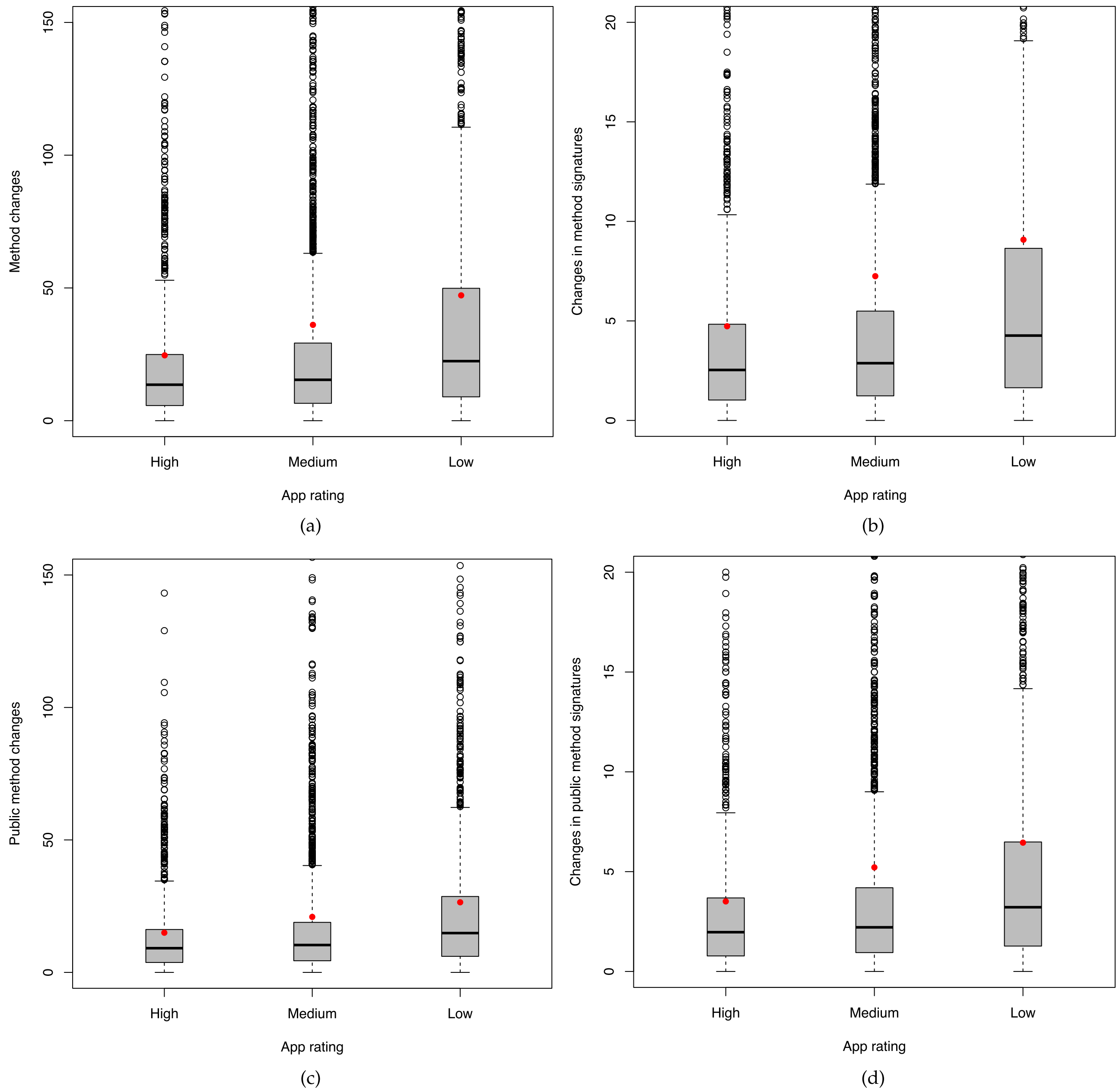


Fig. 4. Boxplots of change-proneness in API classes used by apps having different levels of rating. The red dot indicates the mean.

change-proneness. In particular, the APIs used by apps having a *high* rating underwent, on average, 25 method changes, as opposed to the 36 changes in the APIs used by apps having a *medium* rating (+44 percent) and to the 47 (+88 percent) of the apps having a *low* rating—see Fig. 4a. Also, the three quartiles show a continuous upward-trend of the number of changes as the app ratings decrease.

The trend is almost the same if considering public methods only: an average of 15 method changes for APIs used by top rated apps, 21 for those having a *medium* rating (+40 percent), and 26 for APIs used by apps having a *low* rating (+73 percent)—Fig. 4c. Again, boxplots confirm that apps having a *low* rating generally use more change-prone APIs as compared to apps having a *high* rating.

Also for changes involving method signatures (Figs. 4b and 4d), results highlight that highly rated apps are generally built using stable APIs. If considering both public and private/protected methods (Fig. 4b), we observe, on average, five changes in APIs used by apps having a *high* rating, seven changes for apps having a medium rating (+40 percent), and nine for the apps having the lower ratings (+80 percent). Results are confirmed if considering public methods only (Fig. 4d).

Similarly to the case of bug fixes, we also compared the 50 most and the 50 least successful apps (in terms of their average rating), and the results for the four types of changes are:

1) the overall number of method changes in API methods are, on average, 20 for the most successful and 83 (+315 percent) for the least successful apps;
2) the number of changes in public methods is 12 for the most successful, and 44 (+267 percent) for the least successful apps;
3) changes to method signatures are 4 vs. 16 (+300 percent) considering all methods, and 3 vs. 11 (+266 percent) by considering public methods only.

TABLE 6
Change-Proneness of APIs for Apps Having Different Levels of Rating: Mann-Whitney Test (p-value) and Cliff's delta (d)

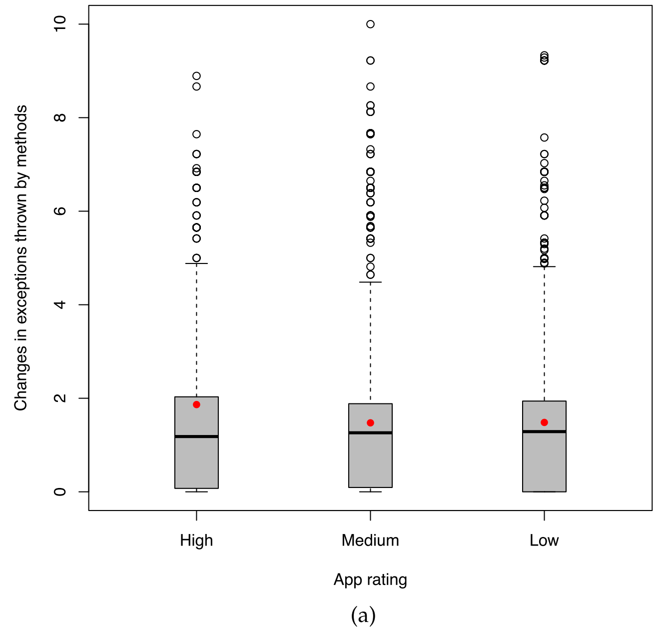
Test	adj. p-value	d
Overall Method Changes		
highrating *vs* medium rating	<0.0001	0.08 (Small)
highrating *vs* low rating	<0.0001	0.25 (Small)
mediumrating *vs* low rating	<0.0001	0.18 (Small)
Changes to Public Methods		
highrating *vs* medium rating	<0.0001	0.08 (Small)
highrating *vs* low rating	<0.0001	0.25 (Small)
mediumrating *vs* low rating	<0.0001	0.17 (Small)
Overall Changes in Method Signatures		
highrating *vs* medium rating	<0.0001	0.07 (Small)
highrating *vs* low rating	<0.0001	0.24 (Small)
mediumrating *vs* low rating	<0.0001	0.17 (Small)
Changes in Public Method Signatures		
highrating *vs* medium rating	<0.0001	0.08 (Small)
highrating *vs* low rating	<0.0001	0.24 (Small)
mediumrating *vs* low rating	<0.0001	0.17 (Small)

Table 6 reports the results of the Mann-Whitney test and the Cliff's d when comparing the change-proneness of APIs used by apps belonging to different groups of average user ratings. Table 6 shows that: (i) there is statistically significant difference (p-value < 0.0001) when comparing apps having a higher rating with those having a lower one, and (ii) Cliff's delta is small for all comparison. However, when comparing the top 50 and the least 50 successful apps (i) the p-value is confirmed < 0.0001, and (ii) we get a *large* Cliff's d (≥ 0.474) for all change types.

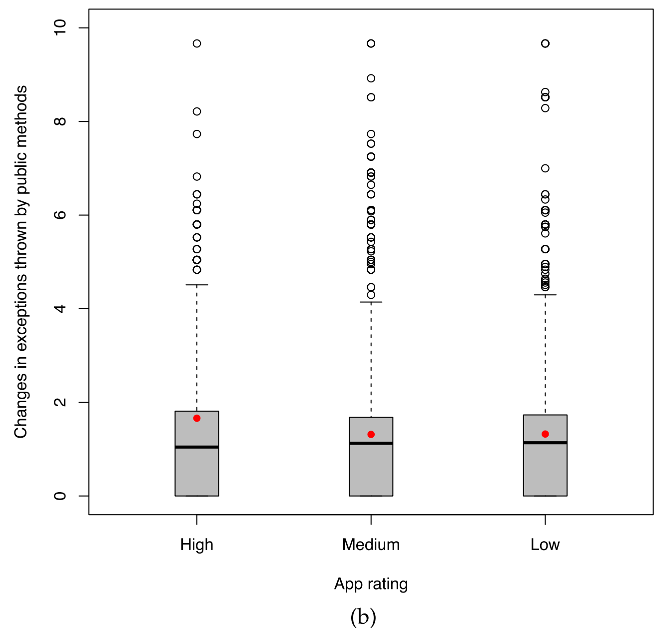
Then, we analyzed another category of changes that might occur in the Android APIs, i.e., changes to the set of exceptions thrown by methods. In total, we identified 2,799 changes to exceptions thrown by methods; 1,735 (62 percent) were aimed at adding new exceptions to a method. Results are reported in Figs. 5a and 5b for all methods and public methods only, respectively. Differently from the trends observed for the other kinds of changes shown in Fig. 4, for what concerns changes to exceptions we do not observe (also according to Mann-Whitney tests performed) any significant difference between different levels of apps' rating. This result is not surprising, since robust Java programs generally make a massive use of exception handling mechanisms [17].

On summary, we can reject our null hypothesis H_{0_2} i.e., *APIs used by apps having high user ratings are on average less prone to changes occurred to API signatures and implementation than APIs used by low rated apps. Instead, there is no significant difference when the changes are on the exceptions thrown by API methods.*

As already done for the fault-proneness, we also analyzed the change-proneness of APIs used by the different categories of apps by isolating official Android APIs and third-party APIs. Concerning the official Android APIs, we observed that those used by apps having high user ratings are significantly less change prone than those used by low rated apps, as also confirmed by the results of the Mann-



(a)



(b)

Fig. 5. Boxplots of changes related to method thrown exceptions in API classes used by apps having different levels of rating. The red dot indicates the mean.

Whitney test reported in Table 7 (p-value always <0.0001 with a small effect size). In particular:

- In terms of overall method changes, apps having a *high* rating use APIs that underwent, on average, 25 changes, as compared to the 37 (+48 percent) of apps having a *medium* rating and the 48 (+92 percent) of apps having a *low* rating. This trend is also confirmed when just considering changes to public methods, with apps having *low* rating using APIs subject to 27 changes, on average, 80 percent more than the apps having *high* rating.
- When focusing on changes performed on method signatures, apps having a *high* rating use APIs object, on average, of 5 changes, 40 percent less than APIs

TABLE 7
Change-Proneness of **Android** APIs for Apps Having Different Levels of Rating: Mann-Whitney Test (p-value) and Cliff's delta (d)

Test	adj. p-value	d
Overall Method Changes		
highrating *vs* medium rating	<0.0001	0.08 (Small)
highrating *vs* low rating	<0.0001	0.26 (Small)
mediumrating *vs* low rating	<0.0001	0.18 (Small)
Changes to Public Methods		
highrating *vs* medium rating	<0.0001	0.08 (Small)
highrating *vs* low rating	<0.0001	0.26 (Small)
mediumrating *vs* low rating	<0.0001	0.18 (Small)
Overall Changes in Method Signatures		
highrating *vs* medium rating	<0.0001	0.07 (Small)
highrating *vs* low rating	<0.0001	0.25 (Small)
mediumrating *vs* low rating	<0.0001	0.18 (Small)
Changes in Public Method Signatures		
highrating *vs* medium rating	<0.0001	0.08 (Small)
highrating *vs* low rating	<0.0001	0.25 (Small)
mediumrating *vs* low rating	<0.0001	0.17 (Small)

TABLE 8
Change-Proneness of **Third-Party** APIs for Apps Having Different Levels of Rating: Mann-Whitney Test (p-value) and Cliff's delta (d)

Test	adj. p-value	d
Overall Method Changes		
highrating *vs* medium rating	<0.0001	0.34 (Medium)
highrating *vs* low rating	<0.0001	0.49 (Large)
mediumrating *vs* low rating	0.0001	0.18 (Small)
Changes to Public Methods		
highrating *vs* medium rating	<0.0001	0.34 (Medium)
highrating *vs* low rating	<0.0001	0.48 (Large)
mediumrating *vs* low rating	0.0002	0.17 (Small)
Overall Changes in Method Signatures		
highrating *vs* medium rating	<0.0001	0.31 (Small)
highrating *vs* low rating	<0.0001	0.49 (Large)
mediumrating *vs* low rating	<0.0001	0.19 (Small)
Changes in Public Method Signatures		
highrating *vs* medium rating	<0.0001	0.30 (Small)
highrating *vs* low rating	<0.0001	0.45 (Large)
mediumrating *vs* low rating	0.0003	0.16 (Small)

used by apps having a *medium* rating and 80 percent less than APIs used by apps having a *low* rating. These results are also confirmed when just focusing on public methods.

- If restricting our analysis to the Android APIs only, we do not observe any statistically significant difference in terms of changes performed to the exceptions thrown by methods between the different categories of apps.

Turning to the third-party APIs, the results of the Mann-Whitney test reported in Table 8 show that the change-proneness of APIs used by apps having high user ratings is lower in a statistically significant way. Moreover, when comparing apps having a *high* rating with those having a *low* rating, we obtain a large effect size for all type of changes reported in Table 8. For instance, when considering all changes performed to the API methods, we go from the three changes, on average, of APIs used by apps having a *high* rating to the seven changes (+133 percent) of APIs used by apps having a *low* rating. The same trend has been also observed when (i) just focusing on public methods, and (ii) just considering the changes occurred to (public) methods' signature.

Instead, also in case of third-party APIs, we did not observe any statistically significant difference in terms of changes performed to the exceptions thrown by methods in APIs used by the different categories of apps.

Summarizing, the results of **RQ**$_2$ show that the higher the average rating of the apps, the lower the change-proneness of the APIs they use. This holds when considering all APIs, as well as when restricting our attention to official Android APIs or third-party APIs only. Instead, there is no significant difference when the changes are on the exceptions thrown by API methods. Again, this result holds for all APIs as well as for the official Android APIs and the third-party APIs considered in isolation.

2.2.3 Qualitative Analysis

The quantitative analysis performed to answer our research questions provided us with strong empirical evidence that Android apps having higher rating generally use APIs that are less fault- and change-prone than APIs used by apps having lower rating. We are aware that this is not sufficient to claim causation; consequently, we performed a qualitative analysis to (at least in part) find a rationale of the relation between the use of "problematic APIs" and the low user ratings of some apps.

First, we performed a coarse grained automatic analysis of comments left by users to unsuccessful apps (i.e., apps having an average rating lower than three), for a total of 15,944 comments. The goal of this analysis is just to get an idea of the main reasons behind the users dissatisfaction with low rated apps. In particular, we are interested in understanding if these comments are mostly related to lack of features in the apps (and thus, no relation with the use of fault- and change- prone APIs can be hypothesized), to bugs/unexpected behavior of apps (and thus, a possible relation with the use fault- and change- proneness APIs could exist), or both. To this aim, we extracted from comments the *n-grams* composing them, considering $n \in [1 \dots 4]$.

Fig. 6 reports the 30 most common n-grams we found. As we can notice, the most frequent n-grams are related to problems with the correct working of the app: *does not work*, *crashes*, *update/needs update*, *please fix it*, *not compatible with*, *freezes*, *can't even open it*, *force close*. However, there are also comments that seems linkable to unsatisfactory features offered by the app: *useless*, *lacks*, *annoying*, *boring*. Thus, as expected, bugs/unexpected behavior of apps represent one of the main reasons behind users dissatisfaction with downloaded apps.

The next step to find insights about the relation between the use of fault- and change-prone APIs and the apps user

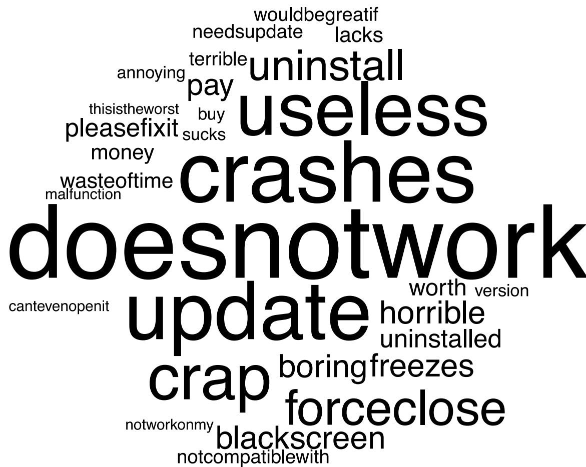


Fig. 6. Word cloud of the 30 most common n-grams in low rated apps user comments.

ratings is to manually analyze some of the unsuccessful apps on Google Play trying to understand if APIs' bugs/frequent changes directly impacted the apps' user experience.

Firstly, we must point out that most of the negative reviews we looked at were simply non-informative, i.e., did not provide any clue for the reasons behind the user dissatisfaction. Examples of such reviews are *"this app is terrible"*, *"crap"*, *"do not download"*, *"improvements needed"*, and *"needs a lot of work"*. This outcome was quite expected, since a recent study by Chen et al. [18] showed that just 35 percent of reviews available on the mobile app marketplace were informative. Also, we found negative reviews due to the poor features provided by the apps (e.g., *"boring"*, *"this is not an app is just a link to the website"*), or to the "spam nature" of the app (e.g., *"a lot of spam on screen and notifications"*, *"I never even got to the point where I could open the app itself I was constantly closing pop-up windows and removing added icons to my home screen"*). These negative reviews are clearly not linkable to any API issues, but simply due to specific apps' characteristics.

Nevertheless, several negative reviews were related to bugs/crashes experienced by users while using the apps (as also highlighted by the n-grams analysis). To provide some numbers, among the 151,564 negative reviews (i.e., those having a score lower than three stars) present in our dataset, 27,162 contained the word "bug" or the trigram "does not work", and 14,228 contained the word "crash", "freezes" or the bi-gram "force close". Most of these reviews did not describe the experienced issues enough in details to allow us to check if the APIs' bugs/frequent changes were the cause of the problem. Still, we found several user reviews directly related to problems present in the APIs used by the apps they downloaded and tried.

An interesting case is the official CNN app for Android tablets. In our study, we analyzed the release 1.3.3 of the CNN app. That version received several low ratings from users (482 out of 812 votes rated the app with one star), mostly because the presence of bugs. However, we found that some of those bugs were related to the Android APIs. For example, these are two reviews in Google Play for the CNN app version 1.3.3

Rating: ★

A Google User - July 3, 2012 - Version 1.3.3
Widget?
The widget looks awesome when it doesn't foul up. I just don't understand the invisible widget thing. please fix.

Rating: ★ ★

A Google User - July 6, 2012 - Version 1.3.3
Needs some MAJOR bug fixes
I was excited to see that the app has finally been updated, and for a few hours it worked great. But then some of its widgets became invisible, and it froze my desktop several times. Galaxy Tab 7.7 with ICS.

By analyzing the change log of the APIs used by the CNN app, we identified a possible cause for the problem described in the reviews. In particular, with a commit performed on 07/03/2012, the developer Chet H. implemented a bug fix solving the issue #6773607 in the Android API: *Layered views animating from offscreen sometimes remain invisible*. The layered views are the mechanism used by the CNN app to implement its widgets.

We also found several user reviews reporting problems related to functionalities in apps that are provided by problematic APIs. An interesting example is the subsystem `android.speech.tts`, providing developers with the possibility of integrating the Text To Speech (TTS) technology in their apps. More than 200 users of the apps using TTS complained about problems related to this feature. Examples of reviews are *"Useless. TTS doesn't work."*, and *"Every time I restart my phone I have to reinstall it as app related to TTS."*. By analyzing the change-history of the `android.speech.tts` subsystem, we found that the 15 classes contained in it underwent, in total, 93 commits (69 of which fixed a bug), distanced on average 13 days from each other. In these commits, a total of 460 methods have been changed, of which 289 are public methods, and among these public methods 80 underwent changes to their signatures. This can suggest that, very likely, it has been difficult, for app developers, to stay tuned with changes performed in such unstable and fault-prone APIs.

Another API, this time a third-party one, that caused problems to users for a certain period of time was the `Facebook Android SDK`.[10] We found almost 100 users of apps relying on the `Facebook Android SDK` leaving low ratings due to problems experienced when logging in, from their app, to Facebook. Examples of these reviews are *"Every time I login to Facebook the app is forced to close."* and *"Started once, seemed to login with Facebook, but after that, it went back to the main screen and nothing happened."* This strange behavior, that forced the apps to close when logging into Facebook, was due to a bug present in the `Facebook Android SDK` until version 3.5. This issue has also been discussed by Android developers in the popular Questions & Answers website `stackoverflow.com`[11] and was resolved in the version 3.5.1 of the `Facebook Android SDK`.

10. http://tinyurl.com/nz7z4zs verified on January 2014.
11. http://tinyurl.com/qyop5q9 verified on January 2014.

In general, the performed qualitative analysis confirmed the results of the quantitative one: *fault- and change-prone APIs represent a serious threat for the success of Android apps*.

3 Study II: Survey with Developers

The *goal* of this study is survey Android developers, with the *purpose* of understanding to what extent they experience problems when using APIs and how much they consider these problems to be related with negative user ratings/ comments. Hence, the study *quality focus* is the developers' perception of the impact change- and fault-prone APIs can have on the apps' user ratings. Such perception insights serve to corroborate the (mainly quantitative) results of the first study where we found a correlation between change- and fault-prone APIs and apps ratings. The *context* of this study consists of 45 professional developers (hereinafter referred to as "participants") providing their opinions about the goals of the study.

3.1 Study Design

In the following, we report the design and planning of the survey study, by detailing the context selection, the research questions, the data collection process, and the analysis method.

3.1.1 Context Selection

As potential participants to this study, we targeted all developers of the apps considered in the first study (Study I). To identify them, we mined the Google play market's webpages of the 5,848 apps considered in our previous study to extract the email address of the related developers. This was possible thanks to the *Contact Developer* field present in each webpage presenting an app on the market. We automatically removed all duplicated e-mail addresses due to multiple apps developed by the same developer(s). This resulted in almost 1,800 e-mail addresses including, of course, those related to customer support (e.g., `ask@`, `support@`, etc). We manually pruned-out these addresses, obtaining in the end 1,221 developers to be contacted. Each developer received an email with instructions on how to participate in our study and a link to the website hosting our survey (details of how data was collected are reported in Section 3.1.4). In the end, we collected 45 responses. Even if this number looks very low, i.e., the response rate is 4 percent (whereas the suggested minimum response rate for survey studies is around 10 percent [19]), we should consider that a number of these developers may be no longer active in the field, might have changed organization (if any, while their emails still being valid), etc. In addition, even if the response rate achieved in our study is quite low, we got a number of responses higher or comparable to similar surveys reported in the literature (e.g., [20], [21]).

3.1.2 Research Questions

This study aims at addressing the following two research questions:

- **RQ$_3$**: *To What Extent Android Developers Experience Problems when Using APIs?* This research question aims at investigating whether Android developers experience problems related to the use of APIs when working on their apps. As done in the Android apps case study (Section 2), we focus our attention on both the official Android APIs as well as third-party APIs.
- **RQ$_4$**: *To What Extent Android Developers Consider Problematic APIs to be the Cause of Negative User Rating/Comments?* This research question aims at investigating whether, from a developer's point-of-view, the usage of problematic APIs negatively impact the apps' ratings.

We answer both research questions by asking Android developers to fill-in a questionnaire we designed.

3.1.3 Survey Questionnaire Design

We designed a survey aimed at collecting developers' opinion needed to answer our two research questions. The study questions are reported in Table 9. For each question, the table specifies whether it is expected an answer in a Likert scale [22] (from 1=very low to 5=very high), a Boolean answer (Yes or No), or an open answer.

The first six questions aimed at gathering information about the background of the developers taking part in our study. In particular, we focus on their experience in mobile development (i.e., number of years of experience, used mobile platforms, and number of apps developed) and on the success of their development activity (i.e., number of downloads and average rating assigned by the users to their apps).

Then, we asked developers about their opinion on the factors negatively impacting apps' user ratings. In particular, we provided participants with four different factors to evaluate (see questions from 7 to 10) providing for each of them an assessment on how much it negatively impacts the user ratings/comments of an app. A score of one means that the factor does not negatively impact an app's rating at all, while a score of five means that the factor has a strong, negative impact on the app rating. The four investigated factors are: (i) the features offered by the app are not useful, (ii) the app is difficult to use, (iii) on the Google play store there are better apps offering the same functionality, and (iv) the presence of bugs/ unexpected behaviors in the app. Note that the latter is the only one on which the use of problematic APIs (both the official as well as the third-party ones) could have some form of impact.

In the third part of the survey (questions from 11 to 15) we asked developers to select the most frequent perceived causes of bugs/crashes in the apps among five possibilities: (i) Java programming errors in the app, (ii) use of third-party libraries affected by bugs, (iii) changes in new releases of third-party libraries, (iv) bugs present in the official Android APIs, and (v) changes in new releases of the Android platform. It is clear that our aim here is to have a first indication about possible problems experienced by developers with APIs when working on their apps. This aspect is investigated more in depth in the next part of our questionnaire: experiences with used APIs. Questions from 16 to 18 ask developers if they ever experienced problems with mobile development APIs and, in this case, to indicate the API name and version.

TABLE 9
Survey Questionnaire Filled in by the Study Participants

Question	Answer
Questions about the developer's background	
1. How many years of experience do you have in Android development?	Open
2. On which other mobile platforms did you develop in the past? (e.g., iOS, BlackBerry, etc.)	Open
3. How many apps have you developed?	Open
4. Please provide URLs for your apps if possible	Open
5. How many times have been downloaded your apps?	Open
6. What is the average rating assigned by users to your apps?	1 | 2 | 3 | 4 | 5
On the factors negatively impacting apps'user ratings (1=very low impact, . . ., 5=very strong impact)	
7. The features offered by the app are not useful	1 | 2 | 3 | 4 | 5
8. The app is difficult to use	1 | 2 | 3 | 4 | 5
9. On the Google play store there are better apps offering the same functionalities	1 | 2 | 3 | 4 | 5
10. Presence of bugs/unexpected behaviors in the app	1 | 2 | 3 | 4 | 5
Select, among the following, the most frequent perceived causes of app bugs/crashes.	
11. Java programming errors in the app	*YES* | *NO*
12. Use of third party libraries affected by bugs (e.g., a bug in a library used by the app causes crashes)	*YES* | *NO*
13. Changes in new releases of third party libraries used by the app cause crashes	*YES* | *NO*
14. Bugs present in the official Android APIs (e.g., a bug in the Android APIs causes the app to crash)	*YES* | *NO*
15. Changes in new releases of the official Android APIs cause the app to crash	*YES* | *NO*
Experiences with used APIs.	
16. Did you ever experience problems with mobile development APIs?	*YES* | *NO*
17. If YES to 16, were they official Android APIs or third party APIs? Indicate release version and describe the problem if possible	Open
18. Did you ever have new bugs in your app due to the new releases of the Android platform?	*YES* | *NO*
Impact of problematic APIs on theuser ratings of your apps.	
19. Did you find evidence about possible relationships between bad user ratings/comments and problems experienced with APIs?	*YES* | *NO*
20. If YES to 19, provide an assessment on the severity of this impact on user bad ratings/comments (1=very low, . . ., 5=very high)	1 | 2 | 3 | 4 | 5
21. If YES to 19, describe examples of problems in APIs that caused issues in your apps, with consequent bad ratings/comments	Open

Finally, the last part of our survey (impact of problematic APIs on the user ratings of your apps-questions from 19 to 21) assesses the impact of problematic APIs on apps' user ratings as experienced by developers.

3.1.4 Data Collection

To automatically collect the answers, the survey was hosted on a Web application named *eSurveyPro*[12] Note that the Web application exploited for our survey allowed developers to complete the questionnaire in multiple rounds, e.g., to answer the first two questions on one day, the others one week later. Developers had 45 days available to respond. At the end of the 45 days we collected 28 complete questionnaires. To enlarge the set of participants in our study, we sent a reminder to the developers that did not answer up to that point and waited for additional 35 days. This allowed us collecting additional 17 questionnaires, leading to a total of 45 completed questionnaires.

3.1.5 Analysis Method

We firstly analyzed, by using descriptive statistics and box plots, the answers provided to the questions related to the developers' background (questions from 1 to 6 in Table 9). The results of this analysis provided us with information about the context in which our study has been performed. Then, to answer $\mathbf{RQ}_3$ we report:

1) Box plots of the answers provided by developers to questions 7-10 (see Table 9), assessing the factors negatively impacting the apps' user ratings. The aim is to verify to what extent the only factor potentially affected by the use of problematic APIs (i.e. the presence of bugs/unexpected behaviors in the app) is felt as important by developers.
2) The percentage of developers indicating change- and fault-prone APIs as one of the most frequent perceived causes of bugs/crashes in their apps (see questions 11-15 in Table 9).
3) The percentage of developers declaring to have experienced problems with mobile development APIs (question 16) and to had bugs in their apps due to new releases of the Android platform (question 18). Also, we present qualitative analysis discussing examples gathered from the developers' answers to question 17 (see Table 9).

Concerning $\mathbf{RQ}_4$, we present (i) the percentage of developers declaring, in question 19, to have observed evidence

12. http://www.esurveyspro.com verified on January 2014.

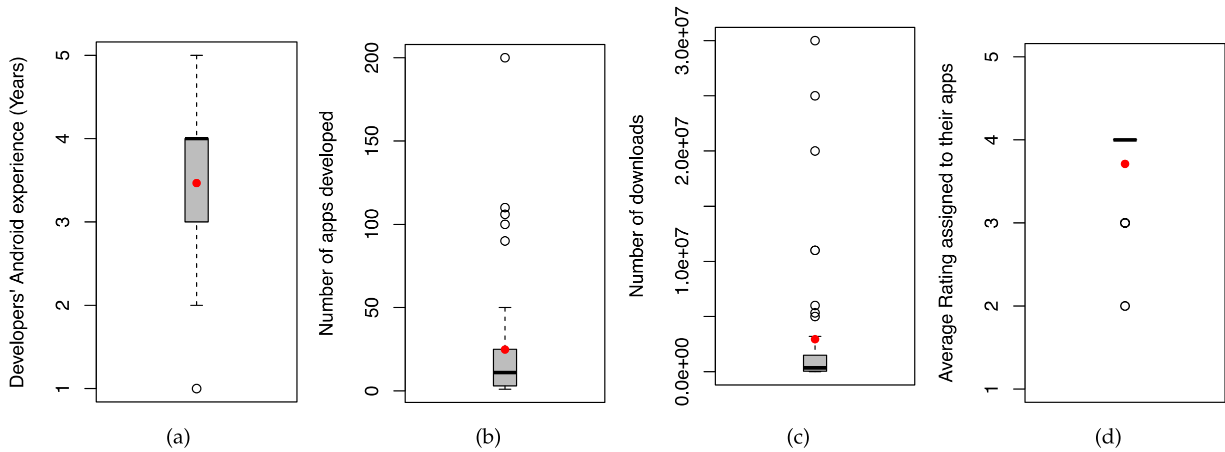


Fig. 7. Boxplots of answers provided by developers to questions related to their experience. The red dots indicate the mean.

about relationships between bad user ratings/comments and problems experienced with mobile development APIs and (ii) box plots of the severity perceived by developers of the negative impact of problematic APIs on user ratings/comments (question 20 in Table 9). Also in this case, we complement our analysis with qualitative data gathered from question 21. Note that questions 20 and 21 were asked only to developers positively answering question 19.

3.1.6 Replication Package

All the data used in our study are publicly available at http://www.cs.wm.edu/semeru/data/tse-android/. Specifically, we provide: (i) the text of the email sent to the developers; (ii) the raw data of answers provided by the developers; (iii) the *R* scripts and working data sets used to run the statistical tests and produce the plots and tables reports in this paper.

3.2 Results

Fig. 7 shows boxplots of the answers provided by participants to questions related to their experience in mobile software development. The 28 developers involved in our study have between two and five years of experience in Android apps development—see Fig. 7a, with a mean of 3.5 years (median 4). They developed between one and 200 apps—see Fig. 7b, with a mean of 25 (median 11), and their apps have been downloaded between 1,932 and 30 millions of times—see Fig. 7c, with a mean of 2,945,000 (median 350,000). The average user ratings of their apps are quite high and inline with what we observed for free apps—see Fig. 7d: the average user rating is included between two and four with an average of 3.7 (median 4).

Overall, the experience of the 45 developers involved in our study is quite high, both in terms of years working on the Android platform (especially considering that Android is a relatively young technology) as well as in terms of number of developed apps. Moreover, 19 of them also developed apps for other mobile platforms, and in particular: 12 developers also worked on iOS, two on PSP, one on NintendoDS, two on BlackBerry, and two on Windows phone. Also, their apps have been downloaded millions of times and, most of them, also received good user ratings.

3.2.1 To What Extent Android Developers Experience Problems when Using APIs?

Fig. 8 reports box plots of the answers provided by developers to questions assessing the negative impact of four different factors (see Table 9—questions 7-10) on the apps' rating. Firstly, it is interesting to note as developers consider the factor less negatively impacting apps' rating the *presence on the Google play market of better apps providing the same functionality*. In fact, this is the factor exhibiting the lowest average score—2.96—and a median of three (i.e., *medium* negative impact on apps' rating). All the other three factors considered in our study exhibited a median score of four (i.e., *strong* negative impact on apps' rating), with the *app's usability* receiving an average score of 3.42, the *uselessness of the features provided by the apps* 4.02, and the *presence of bugs/unexpected behavior* 4.27. Thus, the presence of bugs/unexpected behavior is the factor developers perceived as the one having the strongest negative impact on apps' rating. This is inline with what we observed in the qualitative

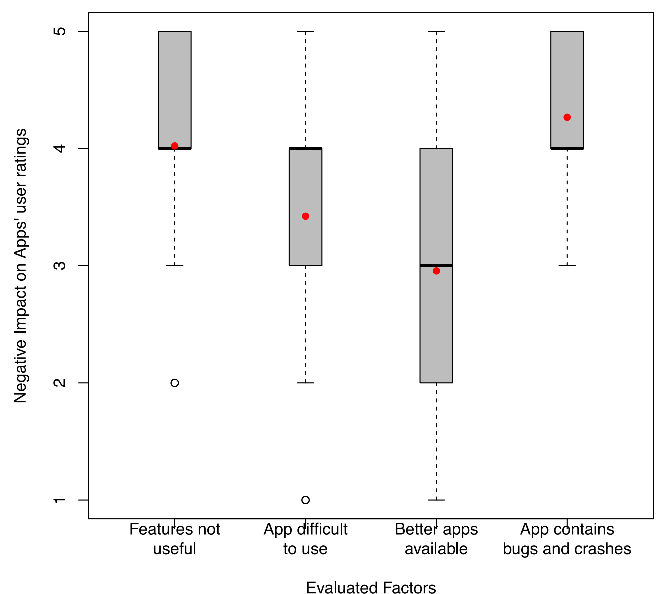


Fig. 8. Boxplots of answers provided by developers to questions 7-10 (see Table 9), assessing the factors negatively impacting the apps' rating (1=very low impact, . . ., 5=very strong impact). The red dots indicate the mean.

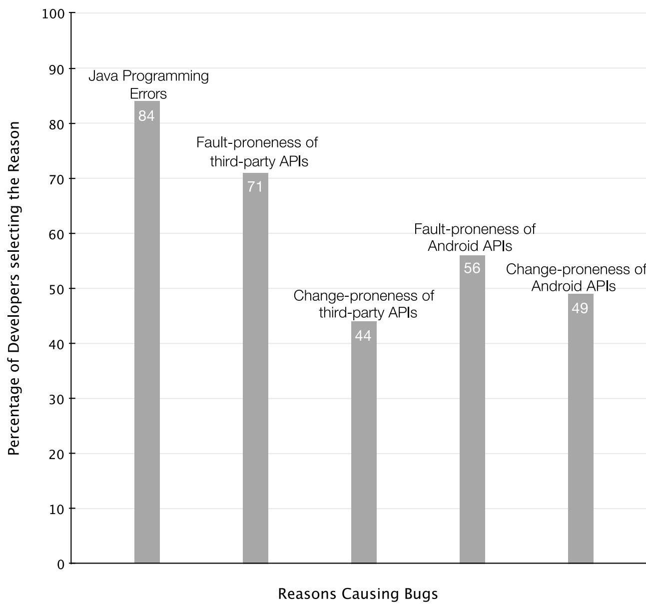


Fig. 9. Percentage of developers indicating each of the considered "perceived causes" among those most frequently causing bugs/crashes in the apps.

analysis performed in the context of our first study (see Section 2.2.3), where we found most of the negative comments left by apps' users related to problems with the correct behavior of the app. Among all factors considered in this study, this is the one having the most straight-forward direct link to API change and fault-proneness. That is, API change- and fault-proneness is unlikely to (directly) affect the app usability or the level of provided functionalities, factors mainly due to app's design and implementation choices (and only partially due to the available technologies). Instead, mis-use of APIs that evolved, or use of unreliable API will likely cause bugs and/or unexpected behavior.

Fig. 9 reports the results obtained when asking developers to select the most frequent perceived causes of bugs/crashes in the apps among: (i) Java programming errors in the app, (ii) use of third-party libraries affected by bugs, (iii) changes in new releases of third party libraries, (iv) bugs present in the official Android APIs, and (v) changes in new releases of the official Android APIs. In particular, for each of these five perceived causes we report the percentage of developers indicating it as one of the most frequent causes of apps' bugs (note that we allowed each developers to select more than one of the proposed causes).

Among the 45 developers, 38 (84 percent) indicated *Java programming errors* as one of the most frequent cause of bugs/crashes in their apps. This result is not surprising since, as any other piece of software, Android apps can be affected by programming errors made by developers. 71 percent of developers (i.e., 32 out of 45) indicated the *use of third-party libraries affected by bugs* as one of the reasons frequently causing bugs/crashes in their apps, while 44 percent (20 out of 45) pointed out the *changes in new releases of third-party libraries* as one of the bugs/crashes root causes. If restricting our attention to the Android official APIs only, 25 developers (56 percent) indicate the *bugs present in the official Android APIs* and 22 (49 percent) the *changes in new releases of the official Android APIs* as frequent cause of bugs/crashes in their apps.

Summarizing, the study results indicate that:

1) a large percentage of the developers (between 44 and 71 percent) consider change- and fault-proneness of APIs as threats to the proper working of their apps. When focusing on problems related to the APIs (i.e., considering all the answers but the "*Java programming errors in the app*" one), developers perceive that bugs present in third-party APIs represent the most frequent cause of bug introduction in their apps.
2) developers are generally more concerned about the effect of bugs present in the used APIs than about changes performed in new releases of the used APIs; this is true for both third-party as well as official Android APIs.
3) developers believe that more bugs are present in third-party APIs than in the official Android APIs. However, they are more concerned about the change-proneness of the Android platform than to the change-proneness of third-party APIs. This result likely has a two-fold explanation. First, the Android APIs have been object of a very fast evolution[13] leading to 18 major releases over just four years. It is very unlikely that also third-party APIs have evolved so fast. This is also confirmed by the average frequency of commits per month observed in Study I for the Android APIs (164 commits per month) as compared to the third-party APIs (14 commits per month). Thus, developers have more likely experienced bugs introduced by major changes in the Android APIs than by changes in the used third-party libraries. Second, Android API reuse by inheritance is widely implemented by developers [23], [24], and Android apps are highly dependent on the official Android APIs [25]. Almost 50 percent of classes in Android apps inherit from a base class as shown in a recent study by Mojica Ruiz et al. [23]. This, again, makes more likely for developers to experience bugs due to changes in the official APIs than in third-party APIs.

Among the 45 developers answering our questionnaire, 33 (73 percent) said they have experienced problems with the used APIs (question 16 in Table 9). Of these 33, 21 indicated Android APIs as the cause of the problems, and 12 indicated third-party APIs. Again, this is likely because most of the APIs used in the apps belong to the Android SDK, and only few of them are third-party ones.[14] Also, 64 percent of developers (29) declared to have observed new bugs in their apps introduced as a consequence of new releases of the Android platform (question 18 in Table 9).

Three developers indicated the third-party library *moPub*[15] as the one they experienced problems with, and one of them also explained the problem. *moPub* is an open-source advertisements (ads) serving platform designed to help developers to monetize the success of their apps by effectively placing advertisements. Note that *moPub* does not broker advertisers for an app; rather, for this task, it

13. https://developer.android.com/reference/android/os/Build.VERSION_CODES.html verified on January 2014.

14. Note that in our first study, we found just 21 percent of the considered apps to use at least one open source third-party API.

15. http://www.mopub.com/ verified on January 2014.

relies on an *ads network*. Hence, *moPub* can be integrated with any available advertisement network, like the one used by the developer, i.e., *MillennialMedia*.[16] The integration between *moPub* and *MillennialMedia* created issues to one of the developers involved in our survey:

> *moPub APIs in some versions caused crashes when integrating MillennialMedia as ad network*

One developer indicated the *google-api-translate-java* APIs[17] as cause of problems in her apps. In particular, while this problem is somewhat related to a third-party API (*google-api-translate-java* is not part of the Android platform), it is manifested just with the release of the Android platform 4.0. The developer pointed us to the *google-api-translate-java* issue tracker describing the problem[18] and wrote:

> *my app makes a massive use of the*google-api-translate-java*APIs and everything worked just fine until the release of Android Ice Cream Sandwich (i.e., the release 4.0 of Android). Then, my app started crashing when invoking the*google-api-translate-java*APIs. The problem was solved by modifying the request to the APIs from a GET to a POST request.*

Other developers indicated some other APIs as the source of their problems (e.g., *RoboGuice*, *Wa*, etc.) without, however, providing a description of the experienced issues.

Summarizing, the quantitative and qualitative results of our $\mathbf{RQ}_3$ highlight that:

1) Developers felt the presence of bugs/unexpected behavior as the main cause of users' bad ratings/comments. Among the factors we investigate, this is the one that has the most direct and straight-forward relationship with the use of problematic APIs.
2) A high percentage of developers (up to 71 percent) consider the change- and fault- proneness of APIs as threats to the proper working of their apps.
3) Seventy three percent of developers experienced problems with the APIs used in their apps. Also, 64 percent declared to have observed new bugs in their apps introduced as a consequence of new releases of the Android platform. These findings have been partially confirmed by the examples described by the developers answering our survey.

3.2.2 To What Extent Android Developers Consider Problematic APIs to be the Cause of Negative User Rating/Comments?

Of the 45 surveyed developers, 28 (62 percent) declared to have observed a relationship between problems experienced with the used APIs and bad user's ratings/comments (question 19 in Table 9). These 28 developers evaluated the severity of the observed impact, providing a score on a five point Likert scale between 1=very low and 5=very high (question 20). Fig. 10 reports the achieved results. The median is 4 (i.e., *high impact*) indicating that the use of problematic APIs could strongly impact the rating of an app form the developers' point-of-view. Also, it is important to note that no one of the developers assessed the impact at a value lower than 3 (i.e., *medium impact*). This means that developers, in their experience, not only observed a decrease of the ratings assigned by users to their apps as consequence of problems in the used APIs, but also that this decrease was substantial.

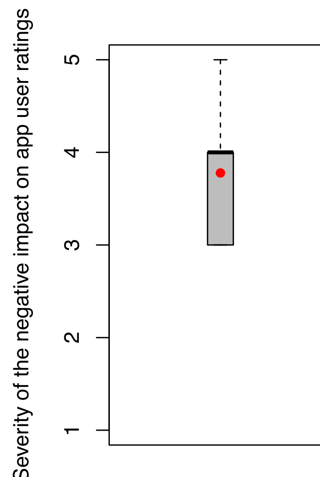


Fig. 10. Severity assigned by developers to the impact of problematic APIs on the rating of their apps (1=very low, ..., 5=very high). The red dot indicates the mean.

Some of the comments left by the developers to question 21 (see Table 9) describe cases where they observed a negative impact of problems experienced with APIs on the ratings/comments left by the apps' users. For instance, one of the developers wrote:

> *my app worked fine until Android 3.2 (API level 13). Then, the app started to crash on screen rotation. This was due to a change in the Android APIs requiring, besides the management of the* `orientation` *value (as needed until API level 12), also the management of the* `screenSize` *value when a screen rotation event arises. Unfortunately, given to commitments on other projects it took some days to fix the problem and this resulted in several low ratings for my apps.*

Another example, reported by two developers, was the removal of the *menu button* that happened with the release of Android Honeycomb (i.e., the release 3.0 of Android). As well explained in a post by Scott Main[19] *Honeycomb removed the reliance on physical buttons, and introduced the* `ActionBar` *class as the standard solution to make actions from the user options immediately visible and quick to invoke*. This change has created several issues to the developers[20] with the need to update their apps as fast as possible. However, as explained by one of the developers involved in our study

> *the removal of the menu button resulted in bad user experiences with my apps and, consequently, in bad user ratings/comments.*

16. http://mmedia.com/ verified on January 2014.
17. https://code.google.com/p/google-api-translate-java/ verified on January 2014.
18. https://code.google.com/p/google-api-translate-java/issues/detail?id=165 verified on January 2014.
19. http://android-developers.blogspot.de/2012/01/say-goodbye-to-menu-button.html verified on January 2014.
20. see e.g., http://tinyurl.com/o95yfty verified on January 2014.

Other developers described situations in which problems in third-party APIs have negatively impacted the app user ratings, like for instance a developer that reported the issue with the *moPub* library described in the context of $\mathbf{RQ}_3$. When commenting the impact of this problem on the rating of her apps, the developer wrote

> *for few days I received bad user comments due to crashes in my app. However, the moPub team rapidly fixed the problem.*

In summary, the answers provided by developers to questions related to $\mathbf{RQ}_4$ indicate that 62 percent of developers perceived a direct relationship between problems experienced with the used APIs and bad users' ratings/comments, and the impact of such APIs on the apps' user ratings was considered as *medium-high*. Also, the discussed examples support the quantitative results obtained in our first study: the use of problematic APIs could represent a threat for the success of Android apps.

4 Threats to Validity

This section describes the threats to validity of both studies presented in Section 2 and Section 3. We discuss such threats together since, as explained in the introduction, Study II has been conducted to provide a rationale to the findings of Study I, i.e., the relation between APIs change- and fault-proneness and the apps' user ratings.

4.1 Construct Validity

Threats to *construct validity* concern the relationship between theory and observation. For Study I, such threats are essentially due to the measurements/estimates on which our study is based. The most important threat is related to using ratings as an indicator of success. We are aware that such ratings can be highly subjective and imprecise. To mitigate such a threat and the randomness/subjectiveness effect, (i) we analyzed a very large sample of apps, and (ii) we discarded apps having less than 10 ratings. Another possibility would have been to use the number of downloads as a mirror for the apps' success. However, we discarded such an option because:

1) Several users just download the app without even installing it, or they immediately uninstall it, because they realize that was not the app they wanted.
2) Mining studies impact the number of apps' downloads. As in our case, we downloaded thousands of apps, but never installed them on devices.
3) In the Google Play market the number of downloads per app is not reported (in fact, none of the mobile markets lists the number of downloads). Google Play just shows the number of app installations in ranges (e.g., from 100,000 to 500,000). Such a number is an aggregated value that includes the number of installs for all the versions of the app. In other words, a user installing the app A_i version 1.0 and then updating A_i to version 1.1, is considered to install it two times. However, such information is not precise enough for the purpose of our study.

One source of imprecision/incompleteness can be related to how we identified the APIs used by the analyzed apps. Although some of the API usages can not be detected when there is no direct invocation (e.g., API calls encapsulated by Java annotations[21]), the *JClassInfo* tool provided us with all the references to Android classes and methods from client-code (i.e., app using the Android SDK). As references we consider (i) direct invocations to Android classes or to methods contained in them, and (ii) dependencies toward classes/interfaces due to inheritance or interface implementations. Thus, we are not capturing cases of overriding, in which the client code is overriding one or more methods from an API; since the client code is providing its own implementation of the method(s), any impact on the app caused by problems (e.g., bugs) in such implementation should not be considered as a responsibility of the API.

Another imprecision/incompleteness can be related to how fault-proneness of APIs is estimated. We chose to consider bug-fixes instead "number of reported bugs" since the latter could represent false alarms. Also, we did not consider dead apps in our study, i.e., apps with inactive development, for which bug-fixes might not be reported. However, we are aware that the information from software repositories can be imprecise/incomplete in terms of the actual number of bug fixes performed on a project [26]. Moreover, our study did not distinguish how the apps used the APIs (e.g., by inheritance or invocation), because the *JClassInfo* tool lists the references between a JAR file and third-party libraries. However, this would not influence our results, because our research questions do not emphasize the relation between change/fault-proneness and a specific type of API usage.

As for Study II, to allow aggregating responses provided by the study participants, wherever appropriate we asked questions using a Likert scale [22]. Where this is not appropriate (e.g., for questions like "Did you ever experience problems with mobile development APIs?") we used Boolean answers; however, in most cases such questions are preliminary to more focused ones for which a Likert scale is used. Instead, questions with open answers are mainly aimed at collecting some qualitative insights from the study participants. Also, in Study II the developers might have been influenced by the questions posed in our survey. For instance, when investigating the causes for app bugs/crashes perceived as most frequent by developers (i.e., questions from 11 to 15 in Table 9), four out of the five options were related to the use of problematic APIs. Several other possible reasons for an app bug/crash were all represented by the "*Java programming errors in the app*" option. However, when designing our questionnaire we focused our attention on reaching a fair compromise between the *quantity of information* gathered and the *time* needed to complete the survey. Indeed, a too long questionnaire could have discouraged developers leading to a lower response rate.

4.2 Conclusion Validity

Threats to *conclusion validity* concern the relationship between treatment and outcome. For Study I, our conclusions are supported by appropriate, non-parametric statistics (p-values were properly adjusted when multiple comparisons were performed). In addition, the practical

21. The Android SDK does not have annotations, but third party libraries can define annotations.

relevance of the observed differences is highlighted by effect size measures.

For Study II the main threat to conclusion validity is the extent to which the set of respondents is representative of the population of developers that worked on the set of applications analyzed in Study I. As explained in Section 3 the response rate of our study is only 4 percent, which is below the response rate often achieved in survey studies [19], i.e. 10 percent. However, explicitly targeting original developers is usually challenging because many of them may not be active, the email addresses are invalid, or even impossible to contact because they are no longer using the e-mail addresses we collected. Also, note that a pool of 45 original developers is above the number of original developers used in many previous studies investigating other software engineering phenomenon, where such a number was between 10 and 14 [27], [28], [29], [30].

4.3 Internal Validity

Threats to *internal validity* concern factors that can affect our results. Most importantly, this work does not claim a cause-effect relation between APIs fault- and change- proneness and the user ratings of apps, which can be due to several other factors. Instead, the purpose of our study is to show that the availability of stable and reliable APIs is important for app developers, and without that the success of produced apps (reflected by the user ratings) can be seriously hindered. In the first study we support such findings with qualitative analysis for which we manually analyzed comments related to ratings.

After that, to provide a justification and plausible explanations to the quantitative findings of Study I, we rely on the quantitative and qualitative information collected by interviewing 45 original developers of the analyzed apps (Study II). However, it should be clear that, although the results and insights collected in Study II provide a meaningful rationale for results of Study I, they cannot directly provide a cause-effect explanation of the specific correlations we have found.

Another possible source of bias for the results of Study I might be the thresholds we used when analyzing the data and presenting our results. We grouped the apps into three levels of rating (i.e., *high*, *medium*, and *low*) based on their average rating (r_a). In particular, apps having r_a lower than the first quartile (bottom 25 percent of the apps) were considered as apps having a *low rating*; apps having r_a between the first and the third quartile (middle 50 percent of apps) were considered as apps having a *medium rating*; apps having r_a higher than the third quartile (top 25 percent of the apps) were considered as apps having a *high rating*. Thus, our thresholds to define the apps' rating categories were based on the quartiles of the distribution of the average rating for the 5,848 considered apps. However, a different choice might lead to different results and, consequently, to different findings. For this reason we performed an additional analysis where we considered different thresholds to group the apps into the three rating categories. In particular, we considered the bottom 33 percent apps (in terms of r_a) as those having a *low rating*; the middle 34 percent apps as those having a *medium rating*; and the top 33 percent apps as those having a *high rating*. Also, we focused our analysis of extreme cases on the 100 most and the 100 least successful apps (instead of the 50 most and 50 least successful apps as done in Section 2). The results were consistent with those discussed in this paper and led to the same findings. Details about this analysis are reported in our replication package.[22]

We also replicated the analysis conducted in Study I isolated to the 1,000 most popular apps in our dataset. This analysis is useful to verify whether it is still possible to observe differences in the change- and fault-proneness of APIs used by apps having different levels of ratings when just considering very popular apps. Since the number of downloads for each app is not available, we used the number of reviews received by an app as a proxy of its popularity. The correlation between the number of downloads and the number of reviews received by an app is something expected (i.e., the more an app is downloaded, the more it is reviewed) and it has been also observed in the recent work by Khalid et al. [31]: "*reviews, [...], are highly correlated with download counts*". Even just focusing on the most 1,000 popular apps, we still observed a correlation between the app success and the change- and fault-proneness of the used APIs. Specifically, the higher the app success the lower the change- and fault-proneness of the APIs it uses. Also for this analysis more details are available in our replication package.

4.4 External Validity

Threats to *external validity* concern the generalization of our findings. We limited our analysis to free apps. It could be the case that our conclusions are no longer valid for paid apps. This is because, for example, users could be more disappointed if they payed for an unreliable poor app, while they may not care that much if a free app occasionally crashes. However, although we could not afford—and could not do for legal reasons—the same kind of study on paid apps, at least we have shown (Section 2, Fig. 2) that the distribution of ratings for free apps and paid apps (a set of randomly selected apps) is comparable.

Although we analyzed a pretty large set of apps belonging to various categories, we are aware that our conclusions may or may not generalize to further apps, and for apps developed for other mobile platforms (e.g., iOS or Windows Mobile).

5 Related Work

The analysis of mobile applications and operating systems has become a hot research topic in the recent years. However, for reasons related to availability of source code and other artifacts (e.g., bugs, change requests, etc.), such studies have been mainly focused on the Android ecosystem. For example, the Mining Challenge track at the 10th Working Conference on Mining Software Repositories (MSR'12) [32] was focused on the analysis of change and bug data in the Android OS. Other studies have been oriented to security issues and malware detection as in [33], [34], [35], [36], [37], [38] and few studies using Android apps have investigated software engineering-related tasks [23], [24], [25], [39], [40], [41], [42], [43], [44].

22. http://www.cs.wm.edu/semeru/data/tse-android/

In this section, we focus our attention on related work concerning empirical studies for evolution- and maintenance-related aspects and analysis of change and bug data in Android applications. We also discuss studies that used changes in APIs to analyze software evolution and stability.

5.1 Empirical Studies Using Android Apps

Several recent works extracted bytecode from APK files, as we did in Study I, to analyze evolution- and maintenance-related aspects in Android apps, such as automatic categorization [39], [40], reuse/cloning and dependencies analysis [23], [24], [25], [41], [44], analysis of development process and Android apps design [42], [43]. Concerning the analysis of Android APIs, only the work by McDonnell et al. [45] is related to ours. However, in the following we describe all those studies to provide the reader with a perspective of the empirical studies that have been done using Android apps.

Shabtai et al. [39] categorized APK files into two root categories of the Android market (i.e., "Games" and "Applications"), using attributes extracted from *dex* files and XML data in the APK files. Sanz et al. [40] used string literals in classes, ratings, application sizes, and permissions to classify 820 applications into several existing categories, such as "Entertainment", "Puzzle and brain games", "Communication", "Multimedia and Video", "Society", "Productivity", and "Tools".

Mojica Ruiz et al. [23], [24] analyzed the extent of code reuse in Android applications. The authors extracted the bytecode of Android apps from APK files to generate class signatures. These latter have been generated by using a technique previously applied by Davies et al. [46], [47] on the Maven Repository. Mojica Ruiz et al. [23], [24] used signatures to compute usage frequencies via inheritance and class reuse. The main conclusion of their studies is that reuse by inheritance and code cloning is prevalent in Android apps. Dresnos [41] also used method signatures to detect similar Android apps, where the signatures included string literals, API calls, exceptions, and control flow structures. Linares-Vásquez et al. [44] analyzed the impact of third-party libraries and obfuscation code when the reuse in Android apps is estimated with the technique by Davies et al. [46], [47].

Syer et al. [25] analyzed dependencies, and source code/churn metrics of three mobile apps (i.e., Wordpress, Google Authenticator, and Facebook SDK) in Android and BlackBerry. The authors analyzed different dimensions of reuse (i.e., by inheritance, interface implementation, API calls) and their main conclusions were that Android apps require less source code but have larger files than in BlackBerry, and depend more on the Android APIs.

Minelli and Lanza [42] proposed a visualization-based analysis for mobile apps using Samoa, which is an interactive tool exploiting historical and structural information from the apps. Although the tool is not focused on a specific design aspect as reuse, the authors used the Average Hierarchy Height (AHH) and Average Number of Derived Classes (ANDC) metrics to study inheritance in Android apps. They found that some apps reuse libraries by copying the entire code instead of referencing JAR files. Some of the findings help to describe the programming model of Android apps (e.g., complexity of mobile apps is mostly attributed to the dependency on third-party libraries), however, only 20 apps were used in the study.

Syer et al. [43] analyzed 15 open source apps to investigate the differences of mobile apps with five desktop/server applications. The comparison was based on two dimensions: the size of the apps and the time to fix defects. The study suggest that mobile apps are similar to UNIX utilities in terms of size of the code and the development team. Also, the findings suggest that mobile app developers are concerned to fixing bugs quickly: over a third of the bugs are fixed within one week and the rest are fixed within one month.

The study by McDonnell et al. [45] is the closest to the one presented in this paper. McDonnell et al. analyzed the evolution of Android APIs (i.e., frequency of changes), and the reaction of client code to API evolution. For the latter purpose, they analyzed 10 open source Android applications from seven domains to investigate into: (i) the degree of dependency on Android APIs; (ii) the lag time between a client API reference and its most recent available version; (iii) the adoption time of new APIs; (iv) the relation between API instability and adoption; and (v) the relationship between API updates and bugs in client code. The results show that client code with more changes to adopt API updates are more prone to bugs; also, fast-evolving APIs are used more, but the time taken for adoption is longer.

Mojica Ruiz et al. [48] also related factors—and specifically the number of Ad (advertisement) libraries—to Android app ratings. They studied 236,245 different apps (236,245 app versions) and found no evidence of relations between the use of Ad libraries and the app rating. However, they found that the use of some specific Ad libraries could negatively affect the app rating. Hence, this is yet another factor that could—in some specific cases as Mojica Ruiz et al. found—influence the rating of apps. As we mentioned in the introduction, our work, as also other related work in this area, does not aim at establishing a cause-effect relationship between one factor (API change- and fault-proneness) and the user ratings of an app, but, rather, to show that there is a correlation and to provide a rational to such quantitative findings through a qualitative analysis of app reviews.

Table 10 lists the number of mobile applications and related categories, that were used in the studies mentioned above. If comparing our study to [23], [24], [25], [39], [40], [41], [42], [43], [45], this is the first study relating the API (Android API and third-party libraries) fault- and change-proneness to the user ratings received by the apps.

5.2 Change and Bug Data Analysis in Android

Martie et al. [49] analyzed discussions in the Android open source project issue tracker, and derived the discussion topic trend and time distributions. Results indicated that (i) Android runtime error was a problematic feature of the Android platform and (ii) the new garbage collector in Android Gingerbread may have resolved issues with the Android runtime and graphics applications that use heavy weight graphics libraries. Although [49] did not investigate the impact of Android platform bugs on Android apps,

TABLE 10
Recent Studies on Analysis of Android Apps, Analyzed Aspects or Purpose, Number of Apps, and Number of Android Categories Covered

Study	Purpose	#apps	#cat.
Shabtai et al. [39]	Apps categorization	2,285	2
Syer et al. [25]	Dependencies analysis	3	NR
Sanz et al. [40]	Apps categorization	820	7
Dresnos [41]	Detection of similar apps	2	1
Mojica Ruiz et al. [23]	Reuse by inheritance and code cloning	4,323	5
Minelli and Lanza [42]	Visualization based analysis	20	NR
Mojica Ruiz *et al.* [24]	Reuse by inheritance and code cloning	$>$ 200K	30
Mojica Ruiz et al. [48]	Use of Ad library and app rating	236K	27
Syer et al. [43]	Size, dependencies and defect fix time	15	NR
McDonnell et al. [45]	API instability and adoption	10	7
Linares-Vásquez et al. [44]	Impact of third-party libraries and obfuscated code in reuse by code cloning	24,379	30
Our study	**Apps user ratings and API change/bug proneness**	**5,848**	**30**

We use NR to distinguish the cases where the number of domain categories is not reported.

it provides empirical evidence of the bugs concerning Android developers and the evolution of the Android API as a reaction to those concerns.

Sinha et al. [50] analyzed the contributions to the Android core code base (AOSP), measuring change activity, contributor density, and industry participation in five AOSP sub-projects (device, kernel, platform, tool-chain, tools). Assaduzzaman et al. [51] mined changes and bug reports in Android to identify changes that introduced the bugs. The links between bugs and changes were identified by looking for keywords in commit messages, and by comparing the textual similarity between the reports and the commit messages.

Our work is different from [49], [50] and [51] for the following two reasons: (i) we computed metrics on bugs and changes in the Android APIs to correlate fault/change proneness with the average user rating of apps, and (ii) we did not analyze textual information in bug reports or commit messages.

5.3 APIs Instability Analysis

Dig and Johnson [52] studied the changes between two major releases of four frameworks and one library written in Java; they found that on average 90 percent of the API breaking changes[23] are refactorings. Hou and Yao [53] analyzed the evolution of AWT/Swing at the package and class level. They found that, during 11 years of the JDK release history, the number of changed elements was relatively small as compared to the size of the whole API, and the majority of them happened in release 1.1. Thus, the main conclusion of their study was that the initial design of the APIs contributes to the smooth evolution of the AWT/Swing API.

Changes in APIs were also studied by Raemaekers et al. [54] to measure the stability of the Apache Commons library. Their findings indicated that a relatively small number of new methods were added in each snapshot to the “Commons Logging” library, while there is more work going on in new methods of “Common Codec” than in old ones.

23. Changes causing an application built with an older version of the component to fail under a newer version.

Mileva et al. [7] analyzed 250 Apache projects to identify usage trends and the popularity of four libraries, and the number of times the projects migrated back to an older version of the libraries; although the purpose of the study is not the analysis of API instability, the findings illustrate how bugs in newer versions of libraries motivate library consumers to switch back to earlier versions. In our study, we did not analyze the developers’ reaction to the instability of fault-proneness of Android APIs (i.e., actions taken as a consequence of the APIs instability and fault-proneness). However, we found some evidence of how the Android APIs instability and fault-proneness has impacted apps quality from the users perspective (e.g., low ratings), and evidence that developers had to adapt quickly their apps as a reaction to the low ratings.

Changes in APIs and frameworks require the adaptation of clients (apps in our case), that can, sometimes, be automated. To this aim, Degenais and Robillard [55] proposed `SemDiff`, a tool to recommend client adaptation required when the used framework evolve. The authors evaluated `SemDiff` on the evolution of the Eclipse-JDT framework and three of its clients. Our study does not aim at investigating how apps can be adapted when APIs change, although the criticality of such changes further support the need for such a kind of adaptation.

Businge et al. [56] analyzed the impact of stable/supported APIs and non supported APIs on survival of Eclipse third-party plugins. Their results show that change proneness of the third party plugins based on non supported APIs is higher, and the fault-proneness of third-party plugins based on stable/supported APIs is lower. Although the quality focus in [56] is the survival of the plugins (in our case we used success of apps in terms of ratings), both studies (ours and [56]) provide evidence on the impact of unstable APIs on the client code using those APIs.

The impact of breaking changes could be a major factor for the development of Android apps in Java, because Android produced significant releases as rapidly as every one to six months. Stability in the Android API is a sensitive and timely topic, given the frequent releases and the number of applications that use these APIs. Similarly to [45], [53], [54], we used the number of changes in

methods as a proxy for change-proneness. Our findings suggest that there is a relation between stability and apps rating: the greater the app rating, the lower the number of changes in methods of Android classes and third-party libraries used in the app.

6 Conclusion and Future Work

This paper investigated the relationship between API change- and fault-proneness and the ratings of Android apps using them. While there is anecdotal evidence that API instability (change-proneness) and fault-proneness may impact the success of software applications, until now there were no rigorous empirical evaluations of such relationships. We filled this gap by performing two studies.

In the first study we estimated the success of 5,848 free Android apps as the average ratings obtained in the Google Play market. Then, we measured fault- and change-proneness of APIs (the official Android APIs as well as the open source third-party APIs) used by those apps. The fault-proneness was measured as the total number of bugs fixed in the used API, while to assess the change-proneness we used the number of changes at method level along three categories: (i) generic changes (including all kinds of changes), (ii) changes applied to method signatures, and (iii) changes applied to the exceptions thrown by methods. Moreover, we performed change-analysis by considering all the methods as well as by just focusing on public methods. Results of this study show that APIs used by apps having high user ratings are significantly less fault-prone than APIs used by low rated apps. In addition, APIs used by highly rated apps are also significantly less change-prone than APIs used by low rated apps, including when changes affected method signatures and especially public methods. Instead, changes to the set of exceptions thrown by methods did not significantly relate with the app rating. These findings hold when considering (i) all the APIs used by apps, (ii) just the official Android APIs used by apps, and (iii) just the open source third-party APIs used by apps.

To provide a quantitative and qualitative explanation to the correlations found in the first study, in the second study we conducted a survey with 45 Android developers. Our questions aimed at investigating potential problems experienced by developers with the use of APIs and their perceived impact on bad user ratings/comments. The quantitative data collected in this study highlight as developers experienced problems caused by the APIs change- and fault-proneness. Moreover, most of them observed a direct relationship between problems experienced with the used APIs and bad users' ratings/comments. The examples discussed by developers also allowed us to further corroborate the findings of our studies.

In summary, although it must be clear that the user ratings of an app—as well as its success—can depend on several factors (e.g., the usage of advertisement libraries [48] or energy consumed by the APIs [57]), whenever possible developers should carefully choose the APIs to be used in their apps: the fault-proneness of APIs can easily be propagated to apps using them, causing crashes or other kinds of failures. Also, a high API change-proneness may trigger the need for frequent app updates that can in turn introduce new bugs. Also, such frequent changes may introduce a behavior that is not expected by apps using the APIs; in other words, APIs may not preserve their back-compatibility. This can either be the cause of bugs in apps using such APIs or, when this does not happen, it may trigger complex changes needed to adapt the current application to the evolved APIs, and this not only can induce bugs, but also it could, in some cases, negatively affect the functional (e.g., feature no longer supported by the API) or non-functional characteristics (e.g., increase of battery consumption, or of CPU/memory usage) of the apps.

While our findings highlight the importance of avoiding change-and fault-prone APIs, it must be clear that selecting the best APIs to use is far from trivial. First, information about the change- and fault-proneness of APIs is currently not available for developers, and they react to API changes looking for answers (related to the changes) in Q&A systems [58]. Developing monitoring systems aimed at providing such information to developers (at least for open source APIs) should be a priority for the research community. In the past, this has been done for example to predict the compatibility of Eclipse plug-in with respect to new Eclipse releases [56], [59]. While extracting information about the fault-proneness of APIs is straightforward (a mining of the issue tracking systems may be sufficient), extracting precise information about the change-proneness requires fine-grained change analysis as done in our study by exploiting the MARKOS Code Analyzer.

Even if considering the information about change- and fault-proneness of APIs as available for developers, avoiding change- and fault-prone APIs might be not obvious. Indeed, sometimes developers need a feature implemented in a specific API, despite its change- and fault-proneness. In these cases, the use of recommendation tools able to identify similar software applications (see for instance the work by McMillan et al. [60], [61], [62], and Moritz et al. [63]) can help developers in looking for alternative APIs, implementing the same features and, hopefully, being less change- and fault-prone. Also, another opportunity would be to integrate API change- and fault-proneness analysis in IDEs code-search mechanisms [64], [65], [66], [67], [68], [69].

Of course, the worst-case scenario may happen as well where, for the specific feature needed by the developer, there are no alternatives but using a change- and/or fault-prone API. In such cases, the developer has to carefully consider the balance between the advantages provided by the features implemented in the API (e.g., saved time/money, reuse of already tested code, etc) and the possible issues derived by its change/fault-proneness. Tools aimed at supporting the developers in evaluating such contrasting goals would be worthwhile in these cases. For instance, these tools could estimate the cost of re-implementing from scratch the feature provided by the API as well as the likelihood of having bugs in the app due to the use of the API.

Lastly, it is possible that app stores could be interested in applying some forms of quality control on the APIs used by the deployed apps, and such quality controls can be built based on the results of this study. However, besides the pros and cons outlined above—including the need for using some APIs when no alternatives are available—this could go against the open philosophy of the app store.

Our future research agenda includes additional studies aimed at further corroborating our results and at empirically investigating other factors impacting the apps' success. Such factors include (i) the change- and fault-proneness of the apps themselves, (ii) the design quality of the apps, and (iii) the responsiveness of developers in implementing features/bug-fixes required by the apps' users. Our work-in-progress also focuses on implementing recommenders to support developers in dealing with APIs updates that can potentially (and inadvertently) impact their apps with breaking changes and bugs, as the ones proposed by Linares-Vásquez [70].

Acknowledgments

The authors would like to thank anonymous FSE'13 and TSE reviewers for their pertinent feedback and useful comments that helped us to improve and steer this work. Also, they are grateful to professional Android developers who participated in our survey. This work was supported in part by the NSF CCF-1016868, NSF CCF-1218129, and NSF CAREER-1253837 grants. Gabriele Bavota and Massimiliano Di Penta were partially supported by the MARKOS project, funded by the European Commission under Contract Number FP7-317743. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors. This paper was an extension of "API Change and Fault Proneness: A Threat to the Success of Android Apps" that appeared in the *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*, Saint Petersburg, Russia, pages 477-487, 2013.

References

[1] VisionMobile. (2013). Developer tools: The foundations of the app economy (developer economics 2013) [Online]. Available: http://www.visionmobile.com/product/developer-economics-2013-the-tools-report/
[2] F. J. Jones, M. J. P. Anson, and F. J. Fabozzi, *The Handbook of Traditional and Alternative Investment Vehicles: Investment Characteristics and Strategies*. Hoboken, NJ, USA: Wiley, 2011.
[3] VisionMobile. (2013). Developer economics q3 2013: State of the developer nation [Online]. Available: http://www.developereconomics.com/reports/q3-2013/
[4] VisionMobile. (2012). The new mobile app economy (developer economics 2012) [Online]. Available: http://www.visionmobile.com/product/developer-economics-2012/
[5] M. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Softw. Eng.*, vol. 16, pp. 703–732, 2012.
[6] M. Zibran, "What makes APIs difficult to use?" *Int. J. Comput. Sci. Netw. Security*, vol. 8, no. 4, pp. 255–261, 2008.
[7] Y. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proc. Joint Int. Annu. ERCIM Workshops Principles Softw. Evol. Softw. Evol. Workshops*, 2009, pp. 57–62.
[8] M. Zibran, F. Eishita, and C. Roy, "Useful, but usable? factors affecting the usability of APIs," in *Proc. 18th Working Conf. Reverse Eng.*, 2011, pp. 151–155.
[9] J. Businge, A. Serebrenik, and M. van de Brand, "Eclipse API usage: The good and the bad," *Softw. Quality J.*, pp. 1–35. (2013). [Online] Available: http://dx.doi.org/10.1007/s11219-013-9221-3
[10] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: A threat to the success of Android apps," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 477–487.
[11] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proc. 19th Int. Conf. Softw. Maintenance*, 2003, pp. 23–32.
[12] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *Proc. 11th Int. Workshop Program Comprehension*, 2003, pp. 134–143.
[13] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. Int. Conf. Softw. Maintenance*, 1996, pp. 244–253.
[14] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Hoboken, NJ, USA: Wiley, 1998.
[15] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian J. Statist.*, vol. 6, pp. 65–70, 1979.
[16] R. J. Grissom and J. J. Kim, *Effect Sizes for Research: A Broad Practical Approach*, 2nd ed. Mahwah, New Jersey, USA: Lawrence Earlbaum Associates, 2005.
[17] M. P. Robillard, and G. C. Murphy, "Designing robust Java programs with exceptions," in *Proc. 8th ACM SIGSOFT Int. Symp. Found. Softw. Eng.: 21st Century Appl.*, 2000, pp. 2–10.
[18] N. Chen, J. Lin, S. Hoi, X. Xiao, and B. Zhang, "AR-Miner: Mining informative reviews for developers from mobile app marketplace," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 767–778.
[19] R. M. Groves, *Survey Methodology*, 2nd ed. Hoboken, NJ, USA: Wiley, 2009.
[20] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 344–353.
[21] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, "Do topics make sense to managers and developers?" *Empirical Softw. Eng.*, pp. 1–37, 2014, http://dx.doi.org/10.1007/s10664-014-9312-1
[22] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. Greenville, SC, USA: Pinter Publishers, 1992.
[23] I. Mojica Ruiz, M. Nagappan, B. Adams, and A. Hassan, "Understanding reuse in the Android market," in *Proc. 20th IEEE Int. Conf. Program Comprehension*, 2012, pp. 113–122.
[24] I. Mojica, B. Adms, M. Nagappan, S. Dienst, T. Berger, and A. Hassan, "A large scale empirical study on software reuse in mobile apps," *IEEE Softw.*, vol. 31, no. 2, pp. 78–86, Mar./Apr. 2014.
[25] D. Syer, B. Adams, Y. Zou, and A. Hassan, "Exploring the development of micro-apps: A case study on the Blackberry and Android platforms," in *Proc. 11th IEEE Int. Working Conf. Source Code Anal. Manipulation*, 2011, pp. 55–64.
[26] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2009, pp. 121–130.
[27] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 671–694, Jul. 2014.
[28] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. D. Lucia, "An empirical study on the developers' perception of software coupling," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 692–701.
[29] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?" in *Proc. 20th ACM SIGSOFT Symp. Found. Softw. Eng.*, 2012, p. 44.
[30] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. de Vries, "Moving into a new software project landscape," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 275–284.
[31] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan, "What do mobile app users complain about? A study on free iOS apps," *IEEE Softw.*, (2014). [Online]. Available: http://dx.doi.org/10.1109/MS.2014.50
[32] E. Shihab, Y. Kamei, and P. Bhattacharya, "Mining challenge 2012: The Android platform," in *Proc. 9th IEEE Working Conf. Mining Softw. Repositories*, 2012, pp. 112–115.
[33] L. Baytuk, M. Herpich, S. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications," in *Proc. 6th Int. Conf. Malicious Unwanted Softw.*, 2011, pp. 66–72.
[34] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for Android malware detection," in *Proc. 7th Int. Conf. Comput. Intell. Security*, 2011, pp. 1011–1015.
[35] T.-E. Wei, C.-H. Mao, A. Heng, H.-M. Lee, H.-T. Wang, and D.-J. Wu, "Android malware detection via a latent network behavior analysis," in *Proc. IEEE 11th Int. Conf. Trust, Security Privacy Comput. Commun.*, 2012, pp. 1251–1258.

[36] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and T. Ronghua, "Analysis of malicious and benign Android applications," in *Proc. 32nd Int. Conf. Distrib. Comput. Syst. Workshops*, 2012, pp. 608–616.

[37] R. Jhonson, W. Zhaohui, C. Gagnon, and A. Stavrou, "Analysis of Android applications' permissions," in *Proc. IEEE 6th Int. Conf. Softw. Security Rel. Companion*, 2012, pp. 45–46.

[38] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 95–109.

[39] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying Android applications using machine learning," in *Proc. Int. Conf. Comput. Intell. Security*, 2010, pp. 329–333.

[40] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. Bringas, "On the automatic categorization of Android applications," in *Proc. IEEE Consumer Commun. Netw. Conf.*, 2012, pp. 149–153.

[41] A. Dresnos, "Android: Static analysis using similarity distance," in *Proc. 45th Hawaii Int. Conf. Syst. Sci.*, 2012, pp. 5394–5403.

[42] R. Minelli and M. Lanza, "Software analytics for mobile applications: Insights and lessons learned," in *Proc. 17th Eur. Conf. Softw. Maintenance Reeng.*, 2013, pp. 144–153.

[43] M. Syer, M. Nagappan, B. Adms, and A. Hassan, "Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source Android apps," in *Proc. Conf. Center Adv. Studies Collaborative Res.*, 2013, pp. 283–297.

[44] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting Android reuse studies in the context of code obfuscation and library usages," in *Proc. 11th IEEE Working Conf. Mining Softw. Repositories*, 2014, pp. 242–251.

[45] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 70–79.

[46] J. Davies, D. M. German, M. W. Godfrey, and A. J. Hindle, "Software bertillonage: Finding the provenance of an entity," in *Proc. IEEE Working Conf. Mining Softw. Repositories*, 2011, pp. 183–192.

[47] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage determining the provenance of software development artifacts," *Empirical Softw. Eng.*, vol. 18, pp. 1195–1237, 2012.

[48] I. Mojica, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan, "Impact of ad libraries on ratings of android mobile apps," *IEEE Softw.*, vol. 31, no. 6, pp. 86–92, Nov./Dec. 2014.

[49] L. Martie, V. Palepu, H. Sajnani, and C. Lopes, "Trendy bugs: Topic trends in the Android bug reports," in *Proc. 9th IEEE Working Conf. Mining Softw. Repositories*, 2012, pp. 120–123.

[50] V. Sinha, S. Mani, and M. Gupta, "Mince: Mining change history of Android project," in *Proc. 9th IEEE Working Conf. Mining Softw. Repositories*, 2012, pp. 132–135.

[51] M. Assaduzzaman, M. Bullock, C. Roy, and K. Schneider, "Bug introducing changes: A case study with Android," in *Proc. 9th IEEE Working Conf. Mining Softw. Repositories*, 2012, pp. 116–119.

[52] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *J. Softw. Maintenance Evol.: Res. Practice*, vol. 18, pp. 83–107, 2006.

[53] D. Hou and X. Yao, "Exploring the intent behind API evolution: A case study," in *Proc. 18th Working Conf. Reverse Eng.*, 2011, pp. 131–140.

[54] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proc. 8th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 378–387.

[55] B. Dagenais, and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 481–490.

[56] J. Businge, A. Serebrenik, and M. van den Brand, "Survival of eclipse third-party plug-ins," in *Proc. Int. Conf. Softw. Maintenance*, 2012, pp. 368–377.

[57] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in Android apps: An empirical study," in *Proc. 11th IEEE Working Conf. Mining Softw. Repositories*, 2014, pp. 2–11.

[58] M. Linares-Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "How do API changes trigger stack overflow discussions? a study on the android SDK," in *Proc. 22nd IEEE Int. Conf. Program Comprehension*, 2014, pp. 83–94.

[59] J. Businge, A. Serebrenik, and M. van den Brand, "Compatibility prediction of Eclipse third-party plug-ins in new Eclipse releases," in *Proc. 12th IEEE Int. Working Conf. Source Code Anal. Manipulation*, 2012, pp. 164–173.

[60] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 364–374.

[61] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1069–1087, Sep./Oct. 2012.

[62] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *Proc. 34th IEEE/ACM Int. Conf. Softw. Eng.*, 2012, pp. 848–858.

[63] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, C. McMillan, M. Grechanik, and M. Gethers, "Export: Detecting and visualizing API usages in large source code repositories," in *Proc. 28th IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2013, pp. 11–15.

[64] D. Cubranic and G. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 408–418.

[65] R. Holmes and A. Begel, "Deep intellisense: A tool for rehydrating evaporated information," in *Proc. Int. Working Conf. Mining Softw. Repositories*, 2008, pp. 23–26.

[66] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Proc. 3rd Workshop Recommendation Syst. Soft. Eng.*, 2012, pp. 85–89.

[67] P. Rigby and M. Robillard, "Discovering essential code elements in informal documentation," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 832–841.

[68] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proc. 3rd Int. Workshop Search-Driven Softw. Develop.*, 2011, pp. 17–20.

[69] M. Rahman, S. Yeasmin, and C. Roy, "Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions," in *Proc. IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng.*, 2014, pp. 194–203.

[70] M. Linares-Vásquez, "Supporting evolution and maintenance of Android apps," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 714–717.

Gabriele Bavota received the PhD degree in computer science from the University of Salerno, Italy, in 2013. He is an assistant professor at the Free University of Bozen-Bolzano, Italy. From January 2013 to October 2014, he has been a research fellow at the University of Sannio, Italy. His research interests include software maintenance, empirical software engineering, mining software repository, refactoring of software systems, and information retrieval. He is the author of more than 50 papers appeared in international journals, conferences, and workshops. He serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSME, MSR, ICPC, SANER, SCAM, and others. He is a member of the IEEE Computer Society.

Carlos Eduardo Bernal-Cárdenas received the BS degree in systems engineering from the Universidad Nacional de Colombia in 2012. He is currently working toward the PhD degree at the College of William and Mary advised by Dr Denys Poshyvanyk. His research interests include software engineering, software evolution and maintenance, information retrieval, software reuse, mining software repositories, mobile applications development, and user experience.

Massimiliano Di Penta is an associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is the author of more than 190 papers appeared in international journals, conferences, and workshops. He serves and has served in the organizing and program committees of more than 100 conferences such as ICSE, FSE, ASE, ICSM, ICPC, GECCO, MSR WCRE, and others. He has been a general cochair of various events, including the 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010), the Second International Symposium on Search-Based Software Engineering (SSBSE 2010), and the 15th Working Conference on Reverse Engineering (WCRE 2008). Also, he has been program chair of events such as the 28th IEEE International Conference on Software Maintenance (ICSM 2012), the 21st IEEE International Conference on Program Comprehension (ICPC 2013), the 9th and 10th Working Conference on Mining Software Repository (MSR 2013 and 2012), the 13th and 14th Working Conference on Reverse Engineering (WCRE 2006 and 2007), the First International Symposium on Search-Based Software Engineering (SSBSE 2009), and other workshops. He is currently a member of the steering committee of ICSME, MSR, SSBSE, and PROMISE. Previously, he has been steering committee member of other conferences, including ICPC, SCAM, and WCRE. He is in the editorial board of *IEEE Transactions on Software Engineering, the Empirical Software Engineering Journal* edited by Springer, and the *Journal of Software: Evolution and Processes* edited by Wiley.

Mario Linares-Vásquez received the BS degree in systems engineering from the Universidad Nacional de Colombia in 2005, and the MS degree in systems engineering and computing from the Universidad Nacional de Colombia in 2009. He is currently working toward the PhD degree at the College of William and Mary advised by Dr. Denys Poshyvanyk, and cofounder of liminal ltda. His research interests include software evolution and maintenance, software reuse, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks. He is a member of the IEEE and ACM.

Rocco Oliveto received the PhD degree in computer science from the University of Salerno, Italy, in 2008. He is an assistant professor in the Department of Bioscience and Territory at the University of Molise, Italy. He is the director of the Laboratory of Informatics and Computational Science of the University of Molise. His research interests include traceability management, information retrieval, software maintenance and evolution, search-based software engineering, and empirical software engineering. He serves and has served as an organizing and program committee member of international conferences in the field of software engineering. In particular, he was the program cochair of TEFSE 2009, the Traceability Challenge chair of TEFSE 2011, the Industrial Track chair of WCRE 2011, the Tool Demo cochair of ICSM 2011, the program cochair of WCRE 2012, and he will be the program cochair of WCRE 2013, SCAM 2014, and ICPC 2015. He is a member of the IEEE Computer Society, ACM, and the IEEE-CS Awards and Recognition Committee.

Denys Poshyvanyk received the MS and MA degrees in computer science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in computer science from Wayne State University in 2008. He is an associate professor at the College of William and Mary in Virginia. He serves as a Program cochair for ICSME'16. He also served as a Program cochair for ICPC'13, WCRE'12, and WCRE'11. His research interests are in software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. He is a member of the IEEE and ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.