API Change and Fault Proneness: A Threat to the Success of Android Apps

Mario Linares-Vásquez[1], Gabriele Bavota[2], Carlos Bernal-Cárdenas[3]
Massimiliano Di Penta[2], Rocco Oliveto[4], Denys Poshyvanyk[1]
[1]The College of William and Mary, Williamsburg, VA, USA
[2]University of Sannio, Benevento, Italy
[3]Universidad Nacional de Colombia, Bogotá, Colombia
[4]University of Molise, Pesche (IS), Italy
mlinarev@cs.wm.edu, gbavota@unisannio.it, cebernalc@unal.edu.co,
dipenta@unisannio.it, rocco.oliveto@unimol.it, denys@cs.wm.edu

ABSTRACT

During the recent years, the market of mobile software applications (apps) has maintained an impressive upward trajectory. Many small and large software development companies invest considerable resources to target available opportunities. As of today, the markets for such devices feature over 850K+ apps for Android and 900K+ for iOS. Availability, cost, functionality, and usability are just some factors that determine the success or lack of success for a given app. Among the other factors, reliability is an important criteria: users easily get frustrated by repeated failures, crashes, and other bugs; hence, abandoning some apps in favor of others.

This paper reports a study analyzing how the fault- and change-proneness of APIs used by 7,097 (free) Android apps relates to applications' lack of success, estimated from user ratings. Results of this study provide important insights into a crucial issue: making heavy use of fault- and change-prone APIs can negatively impact the success of these apps.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement

Keywords

Mining Software Repositories, Empirical Studies, Android, API changes

1. INTRODUCTION

According to a recent study by VisionMobile [27], the mobile handset industry has been growing at 23% Compound Annual Growth Rate (CAGR) in revenues since 2009. 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 [26]. Additionally, the developers' mindshare index during the last three years (2010-2012) shows that iOS and Android are the top two software platforms being used by developers worldwide [26, 27].

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) [19]. Also, APIs not ensuring backward compatibility support are typically hard to use because their instability [28], and API breaking-changes could introduce bugs in 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. For example, Zibran *et al.* [29] 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 (31.1%) of those issues were related to API correctness. *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.*

The goal of this paper is to provide solid empirical evidence about the relation between the success of apps (in terms of user ratings), and the change- and fault-proneness of the underlying APIs. The study has been conducted on

ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia

a set of 7,097 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[1]). Then, we identified the APIs used by those apps, and computed the number of bug fixes that those APIs undergo. 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 estimated success of an app is related to APIs fault-proneness and change-proneness, specifically to different kinds of changes occurring to APIs.

Although there is not always a direct cause-effect relationship between fault- and change-proneness of used APIs and the app's success, our conjecture is that fault- and change-proneness can play an important role in the success, along with the other internal (e.g., app features and usability) or external (e.g., availability of alternative similar apps) factors. That is, a heavy usage of fault-prone APIs can lead to repeated failures or even crashes of the app, hence encouraging the user to give low ratings and possibly even abandoning the app. Similarly, the use of unstable APIs that undergo numerous changes in their interfaces can cause backward compatibility problems or require frequent updates to the app. Such updates, in turn, can introduce defects into the applications using the unstable APIs.

Results of our study demonstrate that Android apps having higher success generally use APIs that are less fault- and change-prone than apps having lower success. For instance, *among 7,097 apps that we analyzed, the 50 least successful apps use APIs that are 500% more fault-prone and 333% more change-prone on average than APIs used by the 50 most successful apps.*

Structure of the paper. Section 2 defines our empirical study and the research questions, and provides details about the data extraction process and analysis method. Section 3 reports the study results, and discusses them from both a quantitative and qualitative point of view. Section 4 discusses the threats that could affect the validity of the results achieved. Section 5 relates this work to the existing literature, while Section 6 concludes the paper and outlines directions for future work.

2. EMPIRICAL STUDY DESIGN

The *goal* of this study is to understand to what extent the APIs fault- and change-proneness could have affected the success of the apps using them. The *context* consists of 7,097 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.

We chose Android apps since Android is the only platform among the top developer-mindshare platforms (i.e., Apple iOS and Google Android) with open-source APIs. Moreover, there are two other factors that make Android APIs a suitable and interesting resource for our empirical study. First, the fast evolution of Android APIs[2] is represented by 17 major releases over four years. Table 1 lists the major releases of the Android APIs, the release times, and the difference in months between each pair of subsequent releases. Given that release periods of Android APIs are short, it is possible that breaking changes appear in packages and classes that are particularly change-prone. The fast evolution could also be explained due to a sheer number of needed hot-fixes. Thus, breaking changes and bugs in APIs could impact the quality of Android apps as perceived by the consumers. Second, reuse is widespread in Android apps, especially when compared to desktop open-source programs. Android API reuse by inheritance is widely implemented by Android developers [17], and Android apps are highly dependent on the APIs [25]. Almost 50% of classes in Android apps inherit from a base class as shown in a recent study by Mojica Ruiz *et al.* [17]. Even though such studies were done on an small set of apps, they still provide valuable insights into the magnitude of software reuse in the Android framework [17, 25].

Table 1: Android API Versions.

Level	Version	Release date	Δ
1	**1.0 Base**	Oct. 2008	-
2	**1.1 Base**	Feb. 2009	4 m.
3	**1.5 Cupcake**	May. 2009	3 m.
4	**1.6 Donut**	Sep. 2009	4 m.
5	**2.0 Eclair**	Nov. 2009	2 m.
6	**2.0.1 Eclair**	Dec. 2009	1 m.
7	**2.1 Eclair**	Jan. 2010	1 m.
8	**2.2 Froyo**	Jun. 2010	5 m.
9	**2.3 Gingerbread**	Nov. 2010	5 m.
10	**2.3.3 Gingerbread**	Feb. 2011	3 m.
11	**3.0 Honeycomb**	Feb. 2011	-
12	**3.1 Honeycomb**	May 2011	3 m.
13	**3.2 Honeycomb**	Jun. 2011	1 m.
14	**4.0 Ice Cream Sandwich**	Oct. 2011	4 m.
15	**4.0.3 Ice Cream Sandwich**	Dec 2011	2 m.
16	**4.1 Jelly Bean**	Jun. 2012	6 m.
17	**4.2 Jelly Bean**	Nov. 2012	4 m.

Table 2 reports characteristics of the apps that we analyzed. 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). As mentioned, we focused on free apps since information needed to perform our study cannot be retrieved for commercial apps, because their bytecode is not publicly available.

2.1 Research Questions

In the context of our study we formulated the following two research questions:

- **RQ$_1$**: *Does the fault-proneness of APIs affect the success of Android Apps?* This research question aims at investigating if Android apps having lower success make heavier use of fault-prone APIs as compared to apps having higher success. The conjecture is that the usage of fault-prone APIs can cause annoying failures and crashes, and for this reason users give 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 successful and unsuccessful apps.

- **RQ$_2$**: *Does the change-proneness of APIs affect the success of Android Apps?* This research question is

[1] `http://play.google.com` verified on June 2013.

[2] `https://developer.android.com/reference/android/os/Build.VERSION_CODES.html` verified on June 2013.

Table 2: Characteristics of the apps (grouped by category) used in our study.

Category	#apps	Classes	KLOC
Arcade	387	7-566	5-6
Books and reference	153	7-778	1-11
Brain	397	5-572	14-22
Business	170	8-226	4-101
Cards	255	8-633	1-4
Casual	390	6-566	2-6
Comics	14	16-43	1-1
Communication	169	6-11	1-10
Education	366	6-87	1-4
Entertainment	695	2-11	1-20
Finance	197	4-107	2-48
Health and fitness	43	6-104	2-7
Libraries and demo	143	1-310	11-56
Lifestyle	421	2-572	1-3
Media and video	269	5-572	2-8
Medical	5	13-107	2-21
Music and audio	282	2-190	3-53
News and magazines	228	5-280	6-64
Personalization	543	4-29	1-23
Photography	237	7-1,974	35-132
Productivity	162	7-217	4-7
Racing	273	15-280	6-48
Shopping	57	5-114	2-23
Social	58	9-318	6-7
Sports	227	7-280	5-6
Sports games	280	6-572	14-20
Tools	539	3-65	1-11
Transportation	34	12-454	8-16
Travel and local	94	8-251	5-77
Weather	9	5-41	1-40
Total	**7,097**	**1-1,974**	**1-101**

similar to $\mathbf{RQ}_1$, however it considers the change-proneness instead of the fault-proneness as the main factor to analyze. Thus, the null hypothesis being tested is:

H_{0_2}: There is no significant difference between the average change-proneness of APIs used by successful and unsuccessful apps.

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

The **independent variable** considered to answer $\mathbf{RQ}_1$ 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 $\mathbf{RQ}_2$ the **independent variables** are the number of changes performed in APIs used by the considered apps. 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 $\mathbf{RQ}_1$, the analysis of changes was performed in the same time period considered for bug fixes.

2.2 Data Extraction Process

The data needed to answer our research questions are (i) the user ratings of the 7,097 considered apps, (ii) the list of Android 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. To prune out unreliable ratings, we only consider apps having at least ten votes. With a smaller number of ratings, there is 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.

To identify APIs used by the apps in our study, we downloaded their APK (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 files to jars using the *dex2jar*[4] disassembler tool. Note that, from the list of apps available in Google Play we selected randomly a list of apps to download. However, a downloaded app was included in the analysis only if its APK could be converted to a JAR file without errors;
2. we extracted calls to API classes from `.class` files, using the JClassInfo[5] tool;
3. we discarded all the API classes outside the `android.*` packages.

In this study, we focus only on the official Android APIs, which are generally used by several applications, whereas future work will also consider the impact of third-party APIs used by the apps. The total number of API classes belonging to the `android.*` packages is 4,816.

Once we collected the list of APIs for each app, we mined the APIs entire change history from their Git[6] repositories[7].

[3] `http://code.google.com/p/android-market-api` verified on June 2013.
[4] `http://code.google.com/p/dex2jar` verified on June 2013.
[5] `http://jclassinfo.sourceforge.net` verified on June 2013.
[6] `http://git-scm.com` verified on June 2013.
[7] `https://android.googlesource.com` verified on June 2013.

We analyzed 35,703 developers' commits performed in a period going from September 2009 to January 2013 for a total of 4,781 bug-fixing activities and 370,180 method's changes. We identified bug-fixing commits activities by using an approach proposed by Fischer *et al.* [10], i.e., by mining regular expressions containing issue IDs and the keyword "fix" in the Git 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[8] to compare the APIs before and after each commit at a fine-grain level. In particular, while the Git 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 *srcML* [3], 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), we use 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 [15].

After having analyzed all 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.

2.3 Analysis Method

To define the analysis method it is important to analyze the distribution of successful and unsuccessful apps in our dataset. Figure 1 reports the distribution of the average ratings assigned by users to these apps. Note that the number of ratings received by each app vary between 10 (the minimum we considered) and 450,889, with a first quartile=33, median=115, third quartile=697, and mean=3,135.

As expected, user ratings are generally very high for free apps as those considered in our study. This is likely due to the fact that user expectations are generally lower for free apps than for the paid ones, and also that disappointment for lack of reliability and functionality may be higher when users spend money. In particular, 3,843 apps (54%) exhibit an average rating greater than 4 stars. Nevertheless, due to quite large corpus of apps considered in our study, we also have 538 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 success (in terms of average user rating).

We grouped the apps in four different groups on the basis of their average user rating. In particular, given r_a the average user rating, the four sets are: (i) apps having $r_a > 4$ (3,843 apps), (ii) apps having $3 < r_a \leq 4$ (2,716), (iii) apps having $2 < r_a \leq 3$ (496), and apps having $r_a \leq 2$ (42). Note that we do not consider apps having $r_a \leq 1$ as a separated group, since only one app falls in it.

The results of our research questions were analyzed through box-plots and the Mann-Whitney test [4]. For the latter, we

[8] `http://markosproject.berlios.de` verified on June 2013.

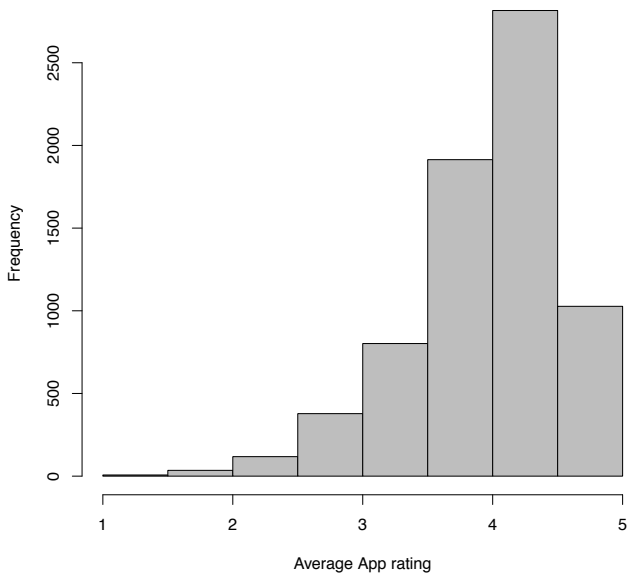


Figure 1: Average user ratings for the 7,097 analyzed apps.

considered two of the four groups of apps at a time, e.g., *apps having* $r_a > 4$ *vs. apps having* $r_a \leq 2$, and we used the Mann-Whitney test to analyze statistical significance of the differences between the fault- and change- proneness of the APIs used by the two groups of apps. The results were intended as statistically significant at $\alpha = 0.05$. Since we performed multiple tests, we adjusted our p-values using the Holm's correction procedure [12]. 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 [11] for ordinal data. We followed the guidelines in [11] 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 $d \geq 0.474$.

2.4 Replication Package

All the data used in our study are publicly available at `http://www.cs.wm.edu/semeru/data/fse-android-api/`. Specifically, we provide: (i) the list (and URLs) of the studied 7,097 apps, together with the user ratings distributions; (ii) the list of Android APIs used by each app; (iii) complete information on the bugs fixed and changes that occurred in the Android APIs; (iv) the *R* scripts and working data sets used to run the statistical tests and produce the plots and tables shown in this paper.

3. ANALYSIS OF THE RESULTS

This section reports the results aimed at answering the two research questions formulated in Section 2.1.

3.1 Fault-Proneness vs. Apps Success

Boxplots in Figure 2 show the distribution of average number of bug fixes in API classes used by apps having different

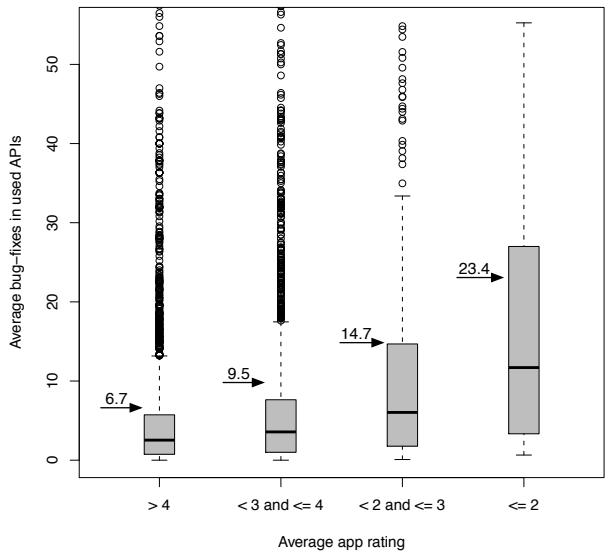


Figure 2: Boxplots of average number of bug fixes in API classes used by apps having different average ratings. The arrow indicates the mean.

average ratings. Note that we set 50 as a limit for the y-axis (i.e., average number of bug fixes in API classes) for readability purposes. As explained in Section 2.3, the apps are grouped into four sets on the basis of their average user ratings (r_a).

The boxplots in Figure 2 highlight that apps having higher user ratings exhibit a lower number of bug fixes in the used APIs. In particular, apps with an average rating greater than four use APIs with 6.7 bug fixes on average. This number increases for apps having lower average ratings: 9.5 (+42%) bug fixes in APIs used by apps with average ratings between 3 and 4; 14.7 (+119%) bug fixes in the set of average ratings between 2 and 3; and 23.4 (+249%) bug fixes in the set of average ratings lower or equal than 2.

Moreover, the distribution of bug fixes (as reported in the boxplots) confirms the conjecture that there is relation between high user ratings and low number of bugs in the APIs that apps use. The difference in terms of API bugs drastically increases when comparing the 50 most and the 50 least successful apps (in terms of achieved average user rating). While for the 50 most successful apps the average number of bug fixes in the used APIs is 4, for the 50 least successful apps we measured an average of 24 bug fixes in the used APIs (+500%).

We also analyzed our data at a finer level of granularity. That is, we computed the average number of bug fixes in a single API class used by the four sets of apps. The results confirm our previous findings: the average number of bug fixes per API class is 0.53 for the most 50 successful apps, and it increases to 2.03 (+217%) for the least 50 successful apps. In particular:

- an API class used by apps having $r_a > 4$ undergoes, on average, 0.69 bug fixes;
- an API class used by apps having $3 < r_a \leq 4$ undergoes, on average, 0.97 bug fixes (+41%);
- an API class used by apps having $2 < r_a \leq 3$ undergoes, on average, 1.33 bug fixes (+93%);
- an API class used by apps having $r_a \leq 2$ undergoes, on average, 1.98 bug fixes (+187%).

Table 3: Use of fault-prone API by apps having different average ratings (r_a): Mann-Whitney test (adj. p-value) and Cliff's Delta (d).

Test	adj. p-value	d
$\mathbf{(r_a > 4)}$ *vs* $(3 < r_a \leq 4)$	<0.0001	0.15 (Small)
$\mathbf{(r_a > 4)}$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.33 (Medium)
$\mathbf{(r_a > 4)}$ *vs* $(r_a \leq 2)$	<0.0001	0.59 (Large)
$\mathbf{(3 < r_a \leq 4)}$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.19 (Small)
$\mathbf{(3 < r_a \leq 4)}$ *vs* $(r_a \leq 2)$	<0.0001	0.49 (Large)
$\mathbf{(2 < r_a \leq 3)}$ *vs* $(r_a \leq 2)$	<0.0001	0.29 (Small)

Table 3 reports the results of the Mann-Whitney test (p-value) and the Cliff's d effect size. We compared each set of apps (grouped by score) with all other sets having a lower average user rating (e.g., $r_a > 4$ vs. the other). As we can notice from the table, apps having a higher average user rating always exhibit a statistically significant lower number of bug fixes in the used APIs than apps having a lower average user rating (p-value always < 0.0001). The Cliff's d is *small* (0.15) when comparing apps having $r_a > 4$ and apps having $3 < r_a \leq 4$, *medium* (0.33) when the comparison is performed between apps having $r_a > 4$ and apps having $2 < r_a \leq 3$, and *large* (0.59) when comparing the top rated apps with those having an average score lower than two. We also observe a *large* d (0.49) when comparing apps having $3 < r_a \leq 4$ with those having $r_a \leq 2$. Thus, there is a strong division between apps having a rating higher than three and those having a rating less than or equal to two.

Summarizing, we can reject our null hypothesis H_{0_1} i.e., *APIs used by successful apps are on average significantly less fault-prone than APIs used by unsuccessful apps.*

3.2 Change-Proneness vs. Apps Success

Boxplots in Figure 3 show the change-proneness of APIs used by the four different sets of apps considered in our study. In particular, Figures 3-(a) and 3-(b) report the overall number of method changes and the overall number of changes in the method signatures, respectively, while Figures 3-(c) and 3-(d) show the same data by considering the APIs' public methods only.

Figure 3 suggests that apps receiving higher average ratings generally use more stable APIs, i.e., APIs having a lower change-proneness. In particular, the APIs used by apps having $r_a > 4$ underwent, on average, 27 method changes, as opposed to the 36 changes in the APIs used by apps having $3 < r_a \leq 4$ (+33%); apps having $2 < r_a \leq 3$ use APIs with 53 method changes (+96%), and apps having $r_a \leq 2$ use APIs with 78 method changes (+189%)—see Figure 3-(a). Also, the three quartiles show a continuous upward-trend of the number of changes as the average rating decreases.

The trend is almost the same if considering public methods only: 16 method changes for APIs used by top rated apps, 21 for the set $3 < r_a \leq 4$ (+31%), 30 for APIs used in apps having $2 < r_a \leq 3$ (+88%), and 41 for APIs in apps

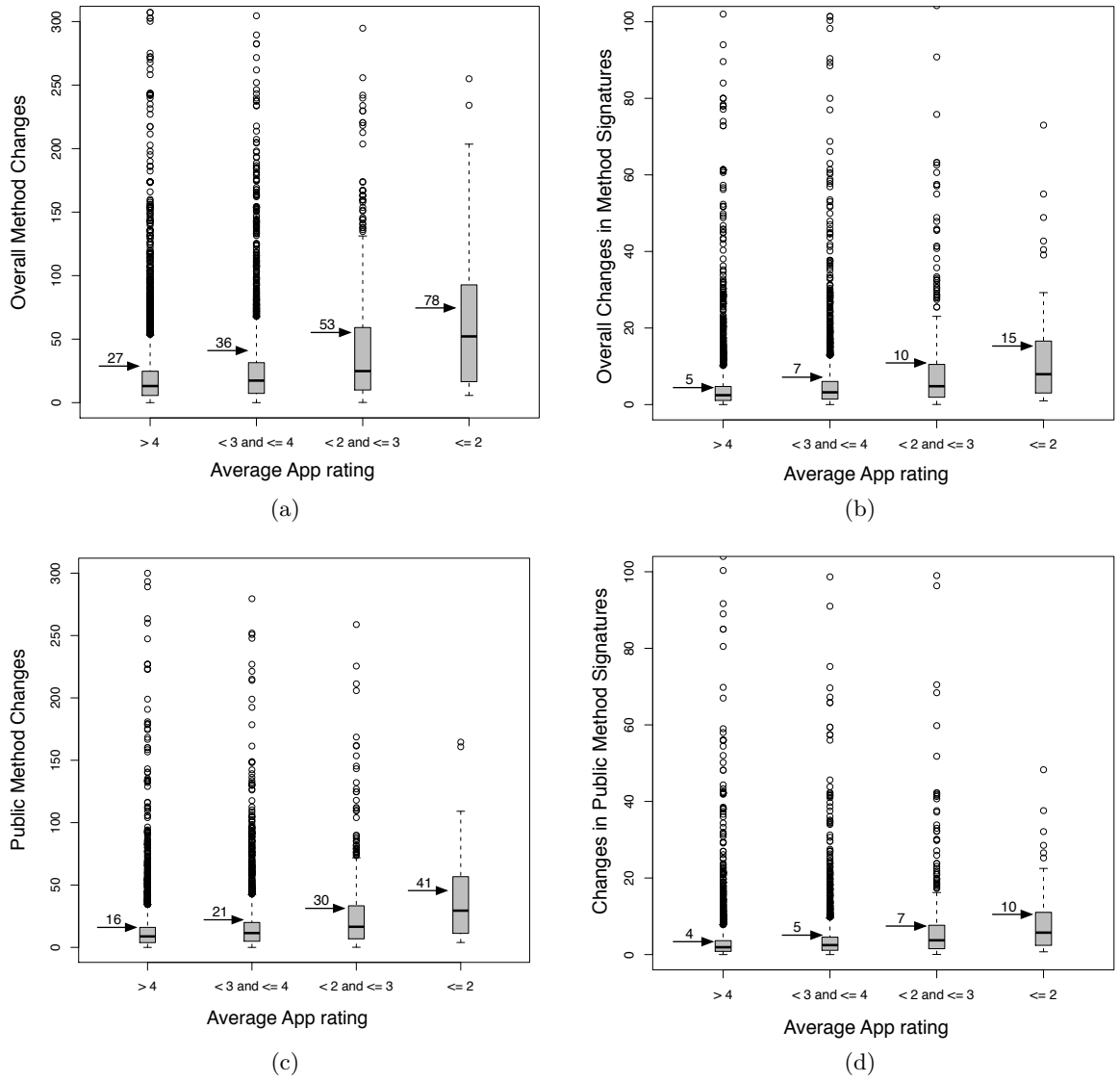


Figure 3: Boxplots of change-proneness in API classes used by apps having different average ratings. The arrow indicates the mean.

having $r_a \leq 2$ (+156%) (Figure 3-(c)). Again, the analysis of boxplots confirms that apps having a low average user rating generally use more change-prone APIs as compared to apps having a high average user rating.

Also for changes involving method signatures (Figure 3-(b,d)), results highlight that successful apps are generally built using stable APIs. If considering both public and private/protected methods (Figure 3-(b)), we observe, on average, five changes in APIs used by apps having an average rating greater than four, 7 changes for apps having $3 < r_a \leq 4$ (+40%), 10 changes for apps having $2 < r_a \leq 3$ (+100%), and 15 for the least successful apps (+200%). Results are confirmed if considering public methods only (Figure 3-(d)).

Moreover, by computing the average number of changes performed in a single API class, the achieved results show that:

- an API class used by apps having $r_a > 4$ underwent, on average, 13 method changes (9 when just focusing on public methods);
- an API class used by apps having $3 < r_a \leq 4$ underwent, on average, 19 method changes, +46% (12 when just focusing on public methods, +33%);
- an API class used by apps having $2 < r_a \leq 3$ underwent, on average, 27 method changes, +108% (16 when just focusing on public methods, +77%);
- an API class used by apps having $r_a \leq 2$ underwent, on average, 41 method changes , +215% (23 when just focusing on public method, +155%).

Similarly to the case of bug fixes, we also compared the 50 most and the 50 least successful apps, and the results for the three types of changes are: (i) the overall number of method changes in API methods are, on average, 18 for the most

Table 4: Change-proneness of APIs for apps having different average rating (r_a): Mann-Whitney test (p-value) and Cliff's delta (d).

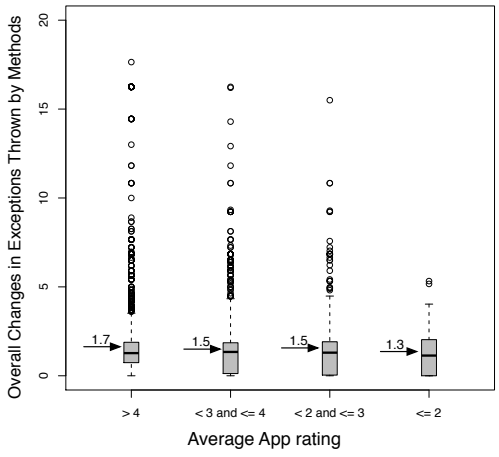
Test	adj. p-value	d
Overall Method Changes		
$(\boldsymbol{r_a > 4})$ *vs* $(3 < r_a \leq 4)$	<0.0001	0.14 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.32 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.57 (Large)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.19 (Small)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.49 (Large)
$(\boldsymbol{2 < r_a \leq 3})$ *vs* $(r_a \leq 2)$	<0.0001	0.28 (Small)
Changes to Public Methods		
$(\boldsymbol{r_a > 4})$ *vs* $(3 < r_a \leq 4)$	<0.0001	0.14 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.32 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.57 (Large)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.19 (Small)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.46 (Large)
$(\boldsymbol{2 < r_a \leq 3})$ *vs* $(r_a \leq 2)$	<0.0001	0.27 (Small)
Overall Changes in Method Signatures		
$(\boldsymbol{r_a > 4})$ *vs* $(3 < r_a \leq 4)$	<0.0001	0.13 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.31 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.58 (Large)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.20 (Small)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.48 (Large)
$(\boldsymbol{2 < r_a \leq 3})$ *vs* $(r_a \leq 2)$	<0.0001	0.28 (Small)
Changes in Public Method Signatures		
$(\boldsymbol{r_a > 4})$ *vs* $(3 < r_a \leq 4)$	<0.0001	0.13 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.31 (Small)
$(\boldsymbol{r_a > 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.58 (Large)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(2 < r_a \leq 3)$	<0.0001	0.19 (Small)
$(\boldsymbol{3 < r_a \leq 4})$ *vs* $(r_a \leq 2)$	<0.0001	0.48 (Large)
$(\boldsymbol{2 < r_a \leq 3})$ *vs* $(r_a \leq 2)$	<0.0001	0.28 (Small)

successful and 78 (+333%) for the least successful apps; (ii) the number of changes in public methods is 11 for the most successful, and 41 (+272%) for the least successful apps; (iii) changes to method signatures are 3 vs. 15 (+400%) considering all methods, and 2 vs. 10 (+400%) if considering public methods only.

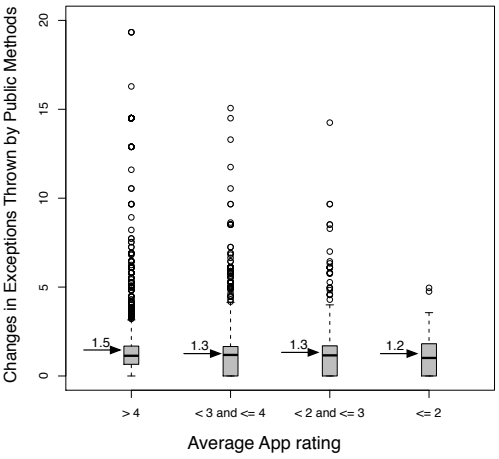
Finally, Table 4 reports the results of the Mann-Whitney test and the Cliff's d when comparing the change-proneness of APIs used by apps belonging to different groups of average user ratings. The main results from Table 4 can be summarized as the following:

- there is statistically significant difference (p-value < 0.0001) when comparing apps having a higher average user rating with those having a lower one.
- we observe a *large* Cliff's delta (≥ 0.474) when comparing the most successful apps (i.e., those having $r_a > 4$) with the less successful ones (i.e., those having $r_a \leq 2$).

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 855 changes to exceptions thrown by methods; 523 (62%) were aimed at adding new exceptions to a method. Results are reported in Figures 4-(a) and 4-(b) for all methods and public methods only, respectively. Differently from the trends observed for the other kinds of changes shown in Figure 3, for what concerns changes to exceptions we do not observe (also according to Mann-Whitney tests performed) any significant difference between different levels of rating. This result is



(a)



(b)

Figure 4: Boxplots of changes related to method thrown exceptions in API classes used by apps having different average ratings. The arrow indicates the mean.

not surprising, since robust Java programs generally make a massive use of exception handling mechanisms [20].

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

3.3 Qualitative Analysis

The quantitative analysis performed to answer our research questions provided us with strong empirical evidence that Android apps having higher success generally use APIs that are less fault- and change-prone than apps having lower success. Although we are aware this is not sufficient to claim causation we performed a qualitative analysis to (at least

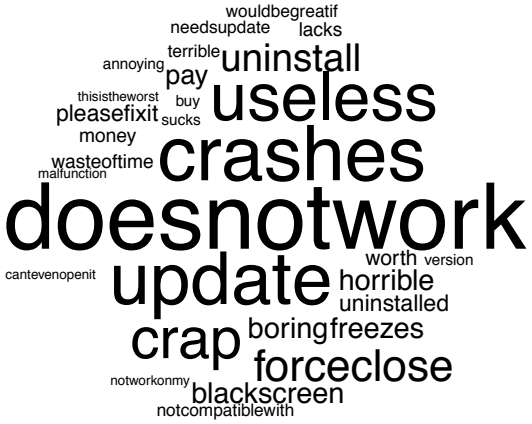


Figure 5: Word cloud of the 30 most common n-grams in unsuccessful apps user comments.

in part) find a rationale of the relation between the use of "problematic APIs" and the low success 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 score 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 unsuccessful 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/malfunctions 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 $n \in [1..4]$.

Figure 5 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/malfunctions of apps represent one of the main reason 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 success 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.

We found several user reviews directly related to problems present in the APIs used by the app that 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 Android 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 were subject, in total, to 93 commits by Android developers, distanced on average 13 days from each other. In these commits, a total of 460 methods have been changed (of which 289 changes to public methods and 80 changes to signatures) and 69 of these changes have been performed to fix bugs. 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.

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.*

4. THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation, and it is 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 ten ratings. Another imprecision/incompleteness can be related to how fault-proneness of APIs is estimated. We chose to consider bug-fixes instead that "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 no 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 [2].

Moreover, our study did not distinguish how the apps used APIs (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.

Threats to *conclusion validity* concern the relationship between treatment and outcome. Our conclusions are supported by appropriate, non-parametric statistics (p-values were properly adjusted when multiple comparisons were performed). In addition, the practical relevance of the observed differences is highlighted by effect size measures.

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 success 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 can be seriously hindered. We support such findings with qualitative analysis for which we manually analyzed comments related to ratings.

Threats to *external validity* concern the generalization of our findings. First, we limited our analysis to free apps. We are aware that the distribution of ratings can be different for commercial apps. 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. Second, although the set of analyzed apps is a small percentage of the existing apps, it is the first time that such number of Android apps is used in an empirical study. Third, as explained in Section 2, we focused on Android internal APIs only; however, the app reliability can also depend on third-party APIs. Finally, our conclusions may not be valid for apps developed for other mobile platforms (e.g., iOS).

5. RELATED WORK

The analysis of mobile applications and operating systems has become a hot research topic in the recent years. For reasons related to its availability, such studies were mainly related to Android. For example, the Mining Challenge track at the 10th Working Conference on Mining Software Repositories (MSR'12) [23] was focused on analysis of change and bug data in the Android OS. However, most of the papers related to Android are aimed to detect security and privacy leaks, such as malware detection and permissions analysis. In this section, we focus our attention to related work concerning bytecode analysis 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 Bytecode Analysis in Android

As in our study, several recent works extracted bytecode from APK files to analyze evolution- and maintenance-related aspects in Android apps. For instance, categorization of Android applications has been explored using machine-learning techniques [22, 21]. Shabtai *et al.* [22] categorized APK files into two root categories of the Android market ("Games" and "Applications"), using attributes extracted from *dex* files and XML data in the APK files. Sanz *et al.* [21] used string literals in classes, ratings, application sizes, and permissions to classify 820 applications into several existing categories.

Table 5: Recent studies on analysis of Android bytecode, analyzed aspects or purpose, number of apps, and number of Android categories covered.

Study	Purpose	#apps	#cat.
Shabtai *et al.* [22]	Apps categorization	2,285	2
Sanz *et al.* [21]	Apps categorization	820	7
Mojica Ruiz *et al.* [17]	Reuse by inheritance and code cloning	4,323	5
Dresnos [9]	Detection of similar apps	2	1
Our study	**Apps success and API change/bug proneness**	**7,097**	**30**

Mojica Ruiz *et al.* [17] analyzed the extent of code reuse in Android applications. The bytecode of Android apps was extracted from APK files to generate class signatures, using a technique that was previously applied by Davies *et al.* [7, 6] on the Maven Repository. Mojica Ruiz *et al.* [17] used signatures to compute usage frequencies via inheritance and class reuse. The main conclusion of their study is that almost 50% of the classes in the apps inherit from a base class, and most of the reused classes are in the Android APIs. Dresnos [9] also used method signatures to detect similar Android apps, where the signatures included string literals, API calls, exceptions, and control flow structures.

Table 5 lists the number of APK files and related categories, that were used in the studies mentioned above. If comparing our study to [22, 21, 17, 9], we claim this is the first time that such a large number of APK files (7,097) has been analyzed, covering 30 domain categories in the official Android market. To the best of our knowledge, this is the first study relating the API fault- and change-proneness to the success of apps.

5.2 Change and Bug Data Analysis in Android

Martie *et al.* [14] 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. Sinha *et al.* [24] 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.* [1] 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 [14], [24] and [1] for the following two reasons: (i) we computed metrics on bugs and changes in the Android APIs to correlate fault/change proneness with the success of apps, and (ii) we did not analyze textual information in bug reports or commit messages.

5.3 APIs Instability Analysis

Dig *et al.* [8] studied the changes between two major releases of four frameworks and one library written in Java; they found that on average 90% of the API breaking changes[9] are refactorings. Hou *et al.* [13] analyzed the evolution of AWT/Swing at the package and class level. Hou *et al.* [13] 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 1.1. Thus, the main conclusion of the study is that the initial design of the APIs contributes to the smooth evolution of the AWT/Swing API.

Changes in APIs also were studied by Raemaekers *et al.* [18] 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, and there is more work going on in new methods of "Common Codec" than in old ones. Mileva *et al.* [16] 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.

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 [5] 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.

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 [13, 18], 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 success: the greater the app rating, the lower the number of changes in methods of Android classes used in the app. However, a deeper analysis on the evolution of Android APIs and Android apps is needed to fully explain this phenomenon.

6. CONCLUSION AND FUTURE WORK

There is anecdotal evidence that API instability (change-proneness) and fault-proneness may impact the success of software applications, but there are no rigorous empirical evaluations of such a relationships. From this point of view, this paper is a premier. Specifically, we empirically analyzed the relationship between the success of 7,097 free Android apps and the stability and fault-proneness of the used Android APIs.

We exploited the apps average ratings in Google Play as a measure of their success. To measure fault-proneness, we used the total number of bugs fixed in the used API; for stability (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. We restricted the analysis 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 dismissed by a new version or (ii) we collected the last rating for such an app.

Our findings show that APIs used by successful apps are significantly less fault-prone than APIs used by unsuccessful apps. In addition, APIs used by successful apps are also significantly less change-prone than APIs used by unsuccessful 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 success. Finally, a manual analysis of users comments and API change logs allowed us to found examples providing a qualitative support to such empirical findings. In summary, although it must be clear that the lack of success of an app can depend on several factors, whenever possible developers should carefully choose the APIs to be used in their apps: fault-prone APIs can in turn cause malfunctions or crashes in apps. Also, API changes may trigger the need for frequent app updates that can in turn introduce new bugs and in general affect the apps' functionality.

Our study establishes some foundations for a research line that could provide developers with elements for replicating "successful apps recipes" or avoiding "unsuccessful apps recipes". It is not only about stable or buggy APIs, but also about features implemented in APIs and reuse. Therefore, further studies should analyze the relationship between features expressed by textual information (in Apps reviews and source code) or API calls, and the success of Android apps. Moreover, sentiment analysis and opinion mining on users' reviews could provide more indication about the factors contributing to the apps success.

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

7. ACKNOWLEDGEMENTS

This work is supported in part by the NSF CCF-0916260, NSF CCF-1016868, and NSF CAREER-1253837 grants. Gabriele Bavota and Massimiliano Di Penta are 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.

8. REFERENCES

[1] M. Assaduzzaman, M. Bullock, C. Roy, and K. Schneider. Bug Introducing Changes: A Case Study with Android. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, Zurich, Switzerland, pp. 116–119. IEEE Press, 2012.

[2] 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 *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software*

Engineering, Amsterdam, The Netherlands, pp. 121–130. ACM Press, 2009.

[3] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based Lightweight C++ Fact Extractor. In *Proceedings of the 11th International Workshop on Program Comprehension*, Portland, Oregon, USA, pp. 134–143. IEEE Computer Society Press, 2003.

[4] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.

[5] B. Dagenais and M. P. Robillard. Recommending Adaptive Changes for Framework Evolution. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, pp. 481–490. ACM Press, 2008.

[6] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software Bertillonage Determining the Provenance of Software Development Artifacts. *Empirical Software Engineering Journal*, 2012.

[7] J. Davies, D. M. German, M. W. Godfrey, and A. J. Hindle. Software Bertillonage: Finding the Provenance of an Entity. In *Proceedings of the 8th IEEE Working Conference on Mining Software Repositories*, Honolulu, Hawaii, pp. 183–192. IEEE Press, 2011.

[8] D. Dig and R. Johnson. How Do APIs Evolve? A Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:83–107, 2006.

[9] A. Dresnos. Android : Static Analysis using Similarity Distance. In *Proceedings of the 45th Hawaii International Conference on System Sciences*, pp. 5394–5403, 2012.

[10] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance*, Amsterdam, The Netherlands, pp. 23–32. IEEE Press, 2003.

[11] R. J. Grissom and J. J. Kim. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Earlbaum Associates, 2nd edition, 2005.

[12] S. Holm. A Simple Sequentially Rejective Bonferroni Test Procedure. *Scandinavian Journal on Statistics*, 6:65–70, 1979.

[13] D. Hou and X. Yao. Exploring the Intent Behind API Evolution: A Case Study. In *Proceedings of the 18th Working Conference on Reverse Engineering*, Limerick, Ireland, pp. 131–140. IEEE Press, 2011.

[14] L. Martie, V. Palepu, H. Sajnani, and C. Lopes. Trendy Bugs: Topic Trends in the Android Bug Reports. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, Zurich, Switzerland, pp. 120–123. IEEE Press, 2012.

[15] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System using Metrics. In *Proceedings of the 12th International Conference on Software Maintenance*, , Monterey, CA, USA, pp. 244–253. IEEE Computer Society Press, 1996.

[16] Y. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining Trends of Library Usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops*, Amsterdam, The Netherlands, pp. 57–62. IEEE 2009.

[17] I. Mojica Ruiz, M. Nagappan, B. Adams, and A. Hassan. Understanding Reuse in the Android Market. In *Proceedings of the 20th IEEE International Conference on Program Comprehension*, Passau, Bavaria, Germany, pp. 113–122. IEEE Press 2012.

[18] S. Raemaekers, A. van Deursen, and J. Visser. Measuring Software Library Stability through Historical Version Analysis. In *Proceedings of the 8th IEEE International Conference on Software Maintenance*, Riva del Garda, Trento, Italy, pp. 378–387. IEEE Press 2012.

[19] M. Robillard and R. DeLine. A Field Study of API Learning Obstacles. *Empirical Software Engineering Journal*, 16:703–732, 2012.

[20] M. P. Robillard and G. C. Murphy. Designing Robust Java Programs with Exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, San Diego, CA, USA, pp. 2–10. ACM Press 2000.

[21] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. Bringas. On the Automatic Categorization of Android Applications. In *Proceedings of the 2012 IEEE Consumer Communications and Networking Conference*, Las Vegas, Nevada, USA, pp. 149–153. IEEE Press 2012.

[22] A. Shabtai, Y. Fledel, and Y. Elovici. Automated Static Code Analysis for Classifying Android Applications using Machine Learning. In *Proceedings of the 2010 International Conference on Computational Intelligence and Security*, Nanning, China, pp. 329–333. CPS, 2010.

[23] E. Shihab, Y. Kamei, and P. Bhattacharya. Mining Challenge 2012: The Android Platform. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, Zurich, Switzerland, pp. 112–115. IEEE Press, 2012.

[24] V. Sinha, S. Mani, and M. Gupta. Mince: Mining Change History of Android Project. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, Zurich, Switzerland, pp. 132–135. IEEE Press, 2012.

[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 *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, Williamsburg, VA, USA, pp. 55–64. IEEE Press, 2011.

[26] VisionMobile. The New Mobile App Economy (Developer Economics 2012), 2012.

[27] VisionMobile. Developer Tools: The Foundations of the App Economy (Developer Economics 2013), 2013.

[28] M. Zibran. What Makes APIs Difficult to Use? *International Journal of Computer Science and Network Security*, 8(4):255–261, 2008.

[29] M. Zibran, F. Eishita, and C. Roy. Useful, but Usable? Factors Affecting the Usability of APIs. In *Proceedings of the 18th Working Conference on Reverse Engineering*, Limerick, Ireland, pp. 151–155. IEEE Press, 2011.