2013 17th European Conference on Software Maintenance and Reengineering

An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps

Pamela Bhattacharya Liudmila Ulanova Iulian Neamtiu Sai Charan Koduru
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
Email:{pamelab,lulan001,neamtiu,scharan}@cs.ucr.edu

***Abstract*—Smartphone platforms and applications (apps) have gained tremendous popularity recently. Due to the novelty of the smartphone platform and tools, and the low barrier to entry for app distribution, apps are prone to errors, which affects user experience and requires frequent bug fixes. An essential step towards correcting this situation is understanding the nature of the bugs and bug-fixing processes associated with smartphone platforms and apps. However, prior empirical bug studies have focused mostly on desktop and server applications. Therefore, in this paper, we perform an in-depth empirical study on bugs in the Google Android smartphone platform and 24 widely-used open-source Android apps from diverse categories such as communication, tools, and media. Our analysis has three main thrusts. First, we define several metrics to understand the quality of bug reports and analyze the bug-fix process, including developer involvement. Second, we show how differences in bug life-cycles can affect the bug-fix process. Third, as Android devices carry significant amounts of security-sensitive information, we perform a study of Android security bugs. We found that, although contributor activity in these projects is generally high, developer involvement decreases in some projects; similarly, while bug-report quality is high, bug triaging is still a problem. Finally, we observe that in Android apps, security bug reports are of higher quality but get fixed slower than non-security bugs. We believe that the findings of our study could potentially benefit both developers and users of Android apps.**

***Keywords*-smartphone apps; Google Android; bug reports; bug fixing; empirical studies; security bugs**

I. Introduction

Smartphone platforms, such as Android and iOS, and the applications (*apps*) that run on these platforms have gained tremendous popularity recently [1]. The presence of app construction frameworks and rich libraries, as well as easy distribution via online app stores such as Google Play and Apple App Store have significantly lowered the barrier to entry in app development and deployment. However, the low barrier to enter the market means apps (or app updates) are subject to limited scrutiny before dissemination, allowing error-prone apps through and therefore affecting user experience. A first step towards correcting this situation is to understand the nature of bugs and the bug-fixing process associated with smartphone platforms and apps. However, empirical bug studies have so far focused mostly on desktop and server applications [2]. The open-source nature of the Android smartphone platform and myriad Android apps hosted on Google Code provide an opportunity to conduct empirical studies and provide a quantitative basis for improving the quality of open-source Android apps. To this end, in this paper, we perform an in-depth empirical study on bugs in the Android Platform project and 24 widely-used open-source Android apps from diverse categories such as communication, tools, media and productivity (Section II). Our analysis has three main thrusts—we chose these thrusts to provide a good breadth–depth balance for our study.

First, we define several metrics to understand the quality of bug reports, and analyze the bug-fix process: bug-fix time, bug categories, bug priority, status and the engagement of the community—users, committers, developers—during the bug-fixing process (Section III). One interesting result of our analysis is visualizing the gap between bugs reported and bugs closed over time for all these Android-based apps; the visualization helps us categorize apps into four different kinds. We also juxtapose changes in this gap over time and the number of developers contributing to the project.

Second, we compare the life-cycle of a bug on Google Code with that of a bug on the popular tracker Bugzilla (Section IV). We show that the lack of certain bug report attributes on Google Code significantly affects the bug-fix process, and makes projects hosted there more difficult to be used for data analysis tasks.

Third, as Android devices carry significant amounts of security-sensitive information, we conduct an investigation of the categories of security bugs in these Android-based apps. In fact, when surveyed, users reported security as the number one concern regarding Android phones and apps [3], and user complaints about Android Platform security are very common.[1] We perform an in-depth study of security bugs in our examined Android apps (Section V). We compare the quality of security bug reports with non-security bug reports, and analyze categories of security bugs. We found that, for Android-based apps, the quality of security bug reports is higher compared to non-security bugs, but the bug-fix time for security bugs is also higher compared to non-security bugs; and that community activity increases when a security bug is reported. We also present several prevalent categories of security bugs in Android-based apps.

[1] Android Platform bug report samples complaining about security issues: http://code.google.com/p/android/issues/detail?id=11211, http://code.google.com/p/android/issues/detail?id=10809,http://code.google.com/p/android/issues/detail?id=8686

1534-5351/13 $26.00 © 2013 IEEE
DOI 10.1109/CSMR.2013.23

To the best of our knowledge, this is the first empirical study that analyzes bug reports and the bug-fixing process in open-source Android projects, and provides a quantitative analysis of security bug reports for the Android platform and open source Android apps.

II. App Overview

Selection criteria: We used the Android Platform and 24 popular open source Android-based projects for our study. In Tab. I we report high-level information for each project. We collect the data in column 2 ("category") and column 3 ("number of downloads") from Google Play—the main app distribution site [4]. The "number of ratings" (column 4) and the "number of bug reports" (column 5) are collected from Google Code—the hosting site for all our examined projects. Column 6 shows the time span between the first and last bug reports we considered. We used several criteria for selecting our apps. First, as can be seen in column 2 of Tab. I, we chose apps from a wide range of categories to reduce selection bias. Second, apps have to be used by a large number of users, which we gauged from the number of downloads and number of ratings. As shown in Tab. I, most apps have more than 1,000 ratings (column 3) and more than 100,000 downloads (column 4). Third, we chose apps with at least 200 bug reports (though 19 of the projects, have more than 500 bug reports; e.g., GAOSP). While the bug count might seem low, one should bear in mind the relative novelty of the platform: Android was first released in September 2008.

Description: We now proceed to describing the apps we used in our study; their names are listed in column 1 of Tab. I. Android Platform is a Linux-based software stack for smartphones, primarily developed by Google and distributed as an open source project [5]. As of April 2012, more than 200 million Android devices were in use worldwide [6]. Firefox Mobile (also known as Fennec) is the Mozilla Firefox Web browser for devices such as mobile phones and PDAs [7]; since Mozilla's Bugzilla bug tracker is a large umbrella for many different Mozilla projects, in our study we only examined Bugzilla bugs filed under Firefox for Android. The remaining apps span a wide range of categories, from communication to personalization, travel and media. All apps in this study are hosted on Google Code, except Firefox Mobile which is hosted at Mozilla.

III. Android Bug Characteristics

In this section, we analyze the bug reports and the bug-fixing process of the Android-based apps along several dimensions. First, we measure the quality of bug reports using several metrics (Section III-A). Second, we analyze the distribution of the bug reports based on their current status (Section III-B). Third, we investigate the activity of the community in terms of the number of developers involved in fixing bugs and adding new code (Section III-C); and by looking at factors like how newly-filed bug reports are handled by the contributors in the project (Section III-E). Fourth, we study bug-fix time in each app (Section III-F). Last, we study how bugs are tossed among multiple developers in the Firefox Mobile project (Section III-F).

A. Bug Report Quality

Prior work has shown that bug report quality critically affects software quality and maintenance effort [8]. One primary factor that dominates the quality of bug reports is how well the bug has been described by the bug reporter when the report was filed. Empirical studies on desktop applications such as Mozilla and Apache have shown that bug reports with certain qualities (clear explanations of the defect, steps for reproducing the bug, description that clearly makes a distinction between expected and obtained results) significantly help developers to understand the problem and reduce the bug-fix time [9]. Therefore, to measure Android bug report quality for the projects we considered, we define five metrics:

- $DescriptionLength$, which counts the number of words in the bug description (significant length indicates a high-quality bug report [9]).
- $ReproduceSteps$, which represents the percentage of bug reports that have steps to reproduce the bug in the bug description.
- $OutputDetails$, which represents the percentage of bug reports that contain details of expected output and actual output.
- $AdditionalInfo$, which represents the percentage of bug reports containing additional information about the bug, besides the standard bug report. [2] Examples of additional information include the version of the application the problem appears in, a special input that triggers the bug, etc.
- $AllDetails$ measures the percentage of bugs that have all three details: $ReproduceSteps$, $OutputDetails$, and $AdditionalInfo$.

High values of these metrics indicate high-quality bug reports, i.e., bug *reporters* are effective in helping bug *fixers* understand the bug. In Tab. II, we report the values of all these metrics for each app. We make several observations: first, we found that average $DescriptionLength$ varies significantly, from a low of 93.97 words (WebSMSDroid) to a high of 327.09 words (Notifier). Second, our analysis shows that WiFiTether has the highest percentage (80.94%) of bug reports that contain steps to reproduce the bug in the bug description ($ReproduceSteps$). Third, bugs reported for the Android Platform and Firefox Mobile rarely have steps to reproduce (only 0.159% and 0.017% of the

[2]For projects hosted on Google Code, this additional information is a special tag. However, in Bugzilla there is no such label, hence the entry for Firefox Mobile, which is hosted on Bugzilla, is marked as N/A.

Project	Category	Number of downloads	Number of ratings	Number of bug reports	Time span
Android Platform	Platform	N/A	N/A	24,707	11/2007–1/2012
Firefox Mobile	Communication	5,000,000–10,000,000	4,510	5,746	10/2009–1/2012
CyanogenMod	Personalization	100,000–500,000	2,512	4,709	8/2009–1/2012
K-9	Communication	1,000,000–5,000,000	34,917	4,028	10/2008–1/2012
CSipSimple	Communication	100,000–500,000	4,640	1,535	3/2010–1/2012
Android-WiFiTether	Communication	1,000,000–5,000,000	21,110	1,387	2/2009–1/2012
ZXing	Libraries-Demo	10,000–50,000	77	1,136	11/2007–1/2012
AnkiDroid Flashcards	Education	100,000–500,000	1,597	956	7/2009–1/2012
Sipdroid	Communication	500,000–1,000,000	8,824	955	4/2009–1/2012
SoftKeyboard	Tools	100,000–500,000	7,222	875	5/2009–1/2012
OsmAnd	Travel-Local	100,000–500,000	2,003	840	4/2010–1/2012
My Tracks	Health-Fitness	1,000,000–5,000,000	53,105	761	5/2010–1/2012
JustPictures	Photography	500,000–1,000,000	11,385	720	2/2010–1/2012
WebSMSDroid	Communication	100,000–500,000	2,796	645	10/2009–1/2012
CallMeter3G	Tools	100,000–500,000	2,171	637	10/2009–1/2012
Android XBMC Remote	Media-Video	100,000–500,000	6,875	603	9/2009–1/2012
ConnectBot	Communication	1,000,000–5,000,000	23,843	547	10/2008–1/2012
GAOSP	Tools	100–500	3	522	2/2010–1/2012
OpenIntents	Productivity	1,000,000–5,000,000	15,694	504	12/2007–1/2012
Android Notifier	Productivity	100,000–500,000	1,977	465	1/2009–1/2012
TransDroid	Tools	N/A	N/A	374	4/2009–1/2012
Android-ADW Launcher	Productivity	1,000,000–5,000,000	73,571	369	10/2010–1/2012
IMSDroid	Media-Video	100,000–500,000	1,338	324	6/2010–1/2012
OSMdroid	Transportation	5,000–10,000	23	302	2/2009–1/2012
Android SMSPopup	Tools	1,000,000–5,000,000	27,744	293	3/2009–1/2012

Table I
ANDROID-BASED APPS USED IN OUR STUDY.

bugs, respectively). Note that the steps to reproduce a bug are not mandatory but have the potential to increase the chances of the bug being fixed quickly. Android-WiFiTether has the highest percentage (74.036%) of bug reports that contain information about the expected and actual output ($OutputDetails$). To summarize, we found that bug reports of most Android-based apps have high quality: bug reporters usually provide (1) long textual descriptions of the problem, (2) steps to reproduce the bug, and (3) explanation of the difference between expected and the actual outputs.

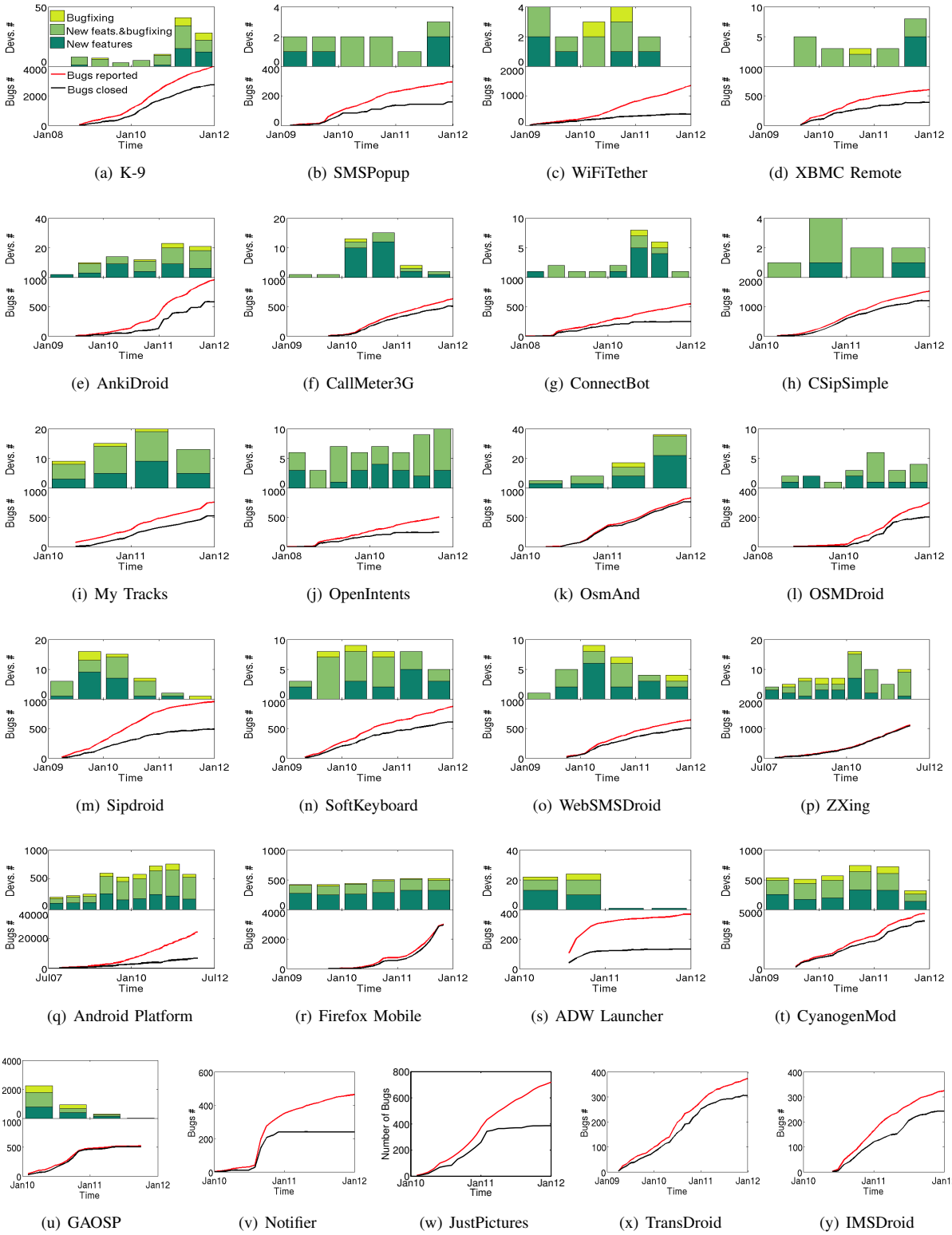
To verify whether highly-rated apps receive high-quality bug reports, we checked whether app ratings correlate with bug quality. That does not seem to be the case: we computed the correlations between number of ratings (Tab. I) and each of the metrics in Tab. II and the correlation values were low (between -0.19 and 0.04).

Comparing bug-fix time for high-quality vs. poor-quality bug reports: To understand the effects of bug report quality on bug-fix time, we performed a regression analysis on the bug report quality metrics. We observed that $DescriptionLength$ is highly correlated with the remaining metrics: $ReproduceSteps$, $OutputDetails$, $AdditionalInfo$ and $AllDetails$ (Pearson's correlation coefficient $0.637 \leq r \leq 0.908$ with p-value < 0.01). Therefore, to understand the effects of bug report quality on bug fix time, we performed a linear regression with $DescriptionLength$ as the single independent variable. We found a negative correlation (coefficient $-0.874 \leq r \leq -0.262$ with p-value < 0.01) for the apps we considered between $DescriptionLength$ and $BugFixTime$. [3] The negative values of the correlation coefficient indicate that $DescriptionLength$ is a good predictor of bug-report quality and that high-quality bug reports get fixed *faster* in our examined apps.

B. Bug Status

As shown in prior empirical analysis on desktop and server applications, bugs have a life-cycle, consisting of changes in bug status, from when a bug is reported to when it is closed [2]. For example, a typical life-cycle of applications using Bugzilla is shown in Fig. 3(a); we will later compare this with the bug life-cycle of Android apps. In this section, we study the distribution of bugs statuses in each app. A high percentage of new bugs in an application is considered a negative indicator of the application's maintenance process [10]. Anecdotal evidence suggests that in addition to affecting software quality, ineffective triaging or poor management of new bugs affect the contributor community, sometimes leading to expert contributors resigning from the community [11]. Therefore, to assess the quality of the triaging process, we also measure how often new bugs are marked fixed, closed, or duplicate.

[3] $BugFixTime = MonthClosed - MonthReported$; the Google Code bug tracker only records the month and year the bug was closed, not the day.



(a) K-9 (b) SMSPopup (c) WiFiTether (d) XBMC Remote

(e) AnkiDroid (f) CallMeter3G (g) ConnectBot (h) CSipSimple

(i) My Tracks (j) OpenIntents (k) OsmAnd (l) OSMDroid

(m) Sipdroid (n) SoftKeyboard (o) WebSMSDroid (p) ZXing

(q) Android Platform (r) Firefox Mobile (s) ADW Launcher (t) CyanogenMod

(u) GAOSP (v) Notifier (w) JustPictures (x) TransDroid (y) IMSDroid

Figure 1. Top graphs show the number of developers involved in bugfixing only, bugfixing and adding new features, or adding new features only. Bottom graphs show bugs reported (top curve, in light color) vs. bugs closed (bottom curve, in black). Section III-C explains the lack of developer data for Notifier, JustPictures, TransDroid, and IMSDroid.

Project	Description Length (words)	Reproduce Steps (%)	Output Details (%)	Additional Info (%)	All Details (%)
Platform	188.37	0.16	0.11	0.04	0.03
ADW Launcher	116.50	31.71	28.99	13.55	10.03
Notifier	327.09	49.46	47.08	9.29	8.86
SMSPopup	128.32	58.36	57.34	31.06	28.67
WiFiTether	153.39	80.94	74.04	46.51	44.36
XBMC Remote	157.69	60.86	58.38	23.38	21.56
AnkiDroid	136.26	32.43	26.05	12.76	11.30
CallMeter3G	95.99	42.02	40.28	15.32	13.90
ConnectBot	169.52	53.38	49.36	25.78	24.50
CSipSimple	152.97	62.92	57.75	38.72	36.49
CyanogenMod	281.16	74.55	0.19	10.04	0.02
Firefox Mobile	323.91	0.02	0.12	N/A	0
GAOSP	153.62	53.46	48.85	11.35	11.35
IMSDroid	201.43	69.44	66.98	31.17	30.56
JustPictures	114.05	36.39	33.33	0	0
K-9	209.78	62.90	49.38	31.73	30.01
My Tracks	157.23	38.13	33.11	9.76	8.97
OpenIntents	165.11	51.59	47.81	25.30	24.10
OsmAnd	140.05	37.88	30.88	9.53	8.81
OSMDroid	161.10	31.56	28.57	14.95	14.29
Sipdroid	200.28	50.37	48.27	28.65	26.23
SoftKeyboard	118.30	42.73	39.98	27.49	26.69
TransDroid	106.75	29.49	0.54	0.27	0.27
WebSMSDroid	93.97	51.47	45.89	19.85	18.92
ZXing	175.38	46.49	43.42	24.05	22.34

Table II
BUG DESCRIPTION METRICS.

In Tab. III we report bug status in the projects we considered. Bug status can span several categories: $Fixed$, $Duplicate$, $Spam$, $Unreproducible$, and $Declined/WontFix$ are self-explanatory; New refers to issues which have not been confirmed as bugs, or have been confirmed as bugs but have not been triaged yet. The category $Others$ refers to bugs kinds such as "need-details," "next release," "future release," "obsolete," "cannotreproduce," "started," or bugs without any status.

We report several observations from our analyses: first, a high percentage of bugs are New in both Android-WiFiTether (70.55%) and the Android Platform (64.98%). Second, OSMDroid and OsmAnd have highest percentage of bugs $Fixed$ (51.82% and 51.39% respectively). Third, Android Platform surprisingly has only 4.97% of the reported bugs fixed and released—we explain why later in this section. Fourth, GAOSP has the highest-percentage of $Spam$ (or invalid) bugs reported (57.16%) while CSipSimple has the highest percentage of $Others$ bugs (40.55%), the majority of which do not have any status. Fifth, for Firefox Mobile alone, we found that 0.011% of the bugs were re-opened—the built-in bug tracker for Google Code does not support the $Reopened$ bug status as Bugzilla does. In Section IV we explain in detail the negative implications of the absence of this status, where bugs cannot be differentiated between statuses New and $Reopened$, because $Reopened$ bugs are marked New after they have been reopened.

C. The Bug Reported–Bug Closed Gap

In addition to considering just the absolute percentage of new bugs for a project, we also study the gap between bugs reported and closed over time, which we name $RCGap$. In Fig. 1, we show the cumulative number of bugs reported and closed [4] over time for the apps. An inspection of the $RCGap$ values reveals four distinct app categories: (1) apps for which $RCGap$ increases significantly over time (Android Platform, Android-SMSPopup, Sipdroid, Android-WiFiTether, Android XBMC Remote, connect-bot, JustPictures, OpenIntents), (2) apps for which $RCGap$ increases at a slower-rate (AnkiDroid Flashcards, CallMeter3G, CSipSimple, IMSDroid, K-9, SoftKeyboard, TransDroid, WebSMSDroid, OSMDroid), (3) apps for which $RCGap$ is constant over time (My Tracks), and (4) apps for which $RCGap$ is close to zero (Firefox Mobile, CyanogenMod, GAOSP, ZXing, OsmAnd). Next, we investigate how developer activity affects this $RCGap$ in these projects; for example, we study if increase or decrease in the number of active contributors affect the $RCGap$ over time.

D. Developer Contribution and $RCGap$

Data collection: To understand how developer participation in these projects affects bug-fixing, we count the number of developers who commit code to the source code repository. We collect developer IDs from commit logs and parse log messages to identify if the commit was due to a bug fix (based on bug fix ID), otherwise we consider the commit to be adding new code. [5]

We report the numbers as stacked bars in Fig. 1 at six-month intervals. [6] The stacked bars indicate: the number of developers adding new features only (bottom bar); the number of developers who are contributing with both fixes and new code (middle bar); the number of developers who only fix bugs (top bar); of course the entire stack indicates the total number of developers contributing to that project in that six-month span.

Analysis: For some apps (K-9Mail, Transdroid) we found that bugs were reported even before the first commit in the source code logs. For most apps, we found that the majority of developers contribute both bug fixes and new code. For apps like WifiTether, CallMeter3G, CSipSimple, K-9Mail, OSMDroid, SipDroid, SoftKeyboard and WebSMSDroid, we found that the value of $RCGap$ increases with the decrease

[4] Closing a bug does not always mean fixing it; it can also refer to marking the bug as duplicate, invalid, or future release. Therefore, the number of bug reports closed is not an indicator of bug-fix time.

[5] We discuss potential threats to validity related to our technique to distinguish between types of commits in Section VI.

[6] We omit developer graphs for four apps. For JustPictures, we did not have access to their code repository. In the case of Notifier and Transdroid, we could not obtain log information for the entire life time of the app. For ImsDroid we did not not have access to the trunk and the logs of the files in branch tags showed commits from a single developer.

in number of committers, indicating lower activity over time. The reverse, however, is not true in the remaining apps: an increase in developer participation does not necessarily lead to a decrease in the $RCGap$. For instance, in apps like SMSPopup, XBMC Remote, and OpenIntents, we see that $RCGap$ increases even with increasing numbers of developers who contribute to the project. For three apps—OsmAnd, Firefox Mobile and ZXing—we observed the expected decrease in $RCGap$ when the number of developers increases.

$RCGap$ in Android Platform*:* We found that, for Android Platform, although the number of developers increases significantly over time, the $RCGap$ increases as well. We investigated the reason behind this ever-increasing gap. We found that Android Platform maintains a private bug database which is not hosted on Google Code. The bug database in Google Code gets a large number of bug reports from users who are different from Android Platform code developers or bug fixers. These bugs are triaged only periodically and most bugs reported in the public database do not go through a typical bug life-cycle process [12]. This unsystematic bug triaging (and consequently the unsystematic bug-fixing process) contributes to the increasing $RCGap$ observed in Android Platform.

In summary, the majority of bug reports have status New and over time the gap between bugs reported and bugs closed increases. This increasing gap can be attributed partly to decreasing developer participation and partly to ineffective triaging.

E. Comment Activity

We measure how responsive and involved a community is by analyzing the bug report comments using two metrics: $FirstComment$, which measures the time it took the first comment to be added to a bug report, and $TotalComments$, which measures the total number of comments on a bug report. We found that the app communities are very responsive, as the median $FirstComment$ time, in days, is 0, and in most cases the average $FirstComment$ is under a day. We found the $TotalComments$ distribution to be skewed, i.e., the median number of comments associated with a bug ranges from 2 to 3, though some bugs receive a disproportionately high number of comments, which shifts the average number of comments' range to 3.52–9.02. We can conclude that the contributor community of the Android-based projects we consider in our study is highly active; users report bugs which are in most cases investigated by at least one contributor and in most cases by more than three contributors. High activity in a project is a metric for effective communication between users and developers in that project and indicates early bug identification.

Project	New (%)	Fixed (%)	Duplicate (%)	Declined (%)	Spam (%)	Others (%)
Platform	64.98	4.97	4.47	11.55	0.66	13.38
ADWLauncher	56.10	7.05	8.67	2.17	11.65	14.36
Notifier	40.82	6.70	21.17	3.24	10.15	17.93
SMSPopup	38.57	16.72	15.36	12.29	9.90	7.17
WifiTether	70.55	5.42	5.56	5.49	7.79	5.19
XBMC Remote	8.62	31.68	11.61	5.14	16.58	26.37
AnkiDroid	10.25	46.76	6.70	1.36	3.24	31.70
CallMeter3G	5.21	22.12	23.07	5.21	36.02	8.37
ConnectBot	46.44	18.28	8.59	4.57	10.79	11.34
CSipSimple	13.21	20.08	14.59	3.92	7.65	40.55
CyanogenMod	6.53	18.63	14.21	4.11	24.16	32.37
Firefox Mobile	29.89	49.11	15.14	0.87	3.79	20.13
GAOSP	1.73	0.19	8.85	4.23	57.12	27.89
IMSDroid	3.41	36.73	5.86	4.94	19.14	29.94
JustPictures	37.78	35.28	7.22	9.44	1.94	8.33
K-9	24.05	22.75	22.30	3.65	9.46	17.79
My Tracks	20.84	25.46	16.10	5.67	4.09	27.84
OpenIntents	41.83	19.92	9.36	4.98	11.75	12.15
OsmAnd	2.29	51.39	7.12	17.37	9.77	12.06
OSMDroid	18.94	51.83	6.65	2.99	5.98	13.62
Sipdroid	30.01	30.85	16.58	0.94	18.78	2.83
SoftKeyboard	21.76	26.12	13.17	2.75	13.17	23.02
TransDroid	10.46	41.02	12.87	9.38	16.35	9.92
WebSMSDroid	9.77	39.38	16.43	4.81	20.47	9.15
ZXing	1.26	35.23	8.11	16.04	13.15	26.22

Table III
BUG STATUS.

F. Bug-fix time

We measured the bug-fix time of each closed bug as $BugFixTime = MonthClosed - MonthReported$. We present the results in Tab. IV. We found that the average time taken to fix a bug for most apps is in the range of 0–1.5 months. We found that AnkiDroid Flashcards and Firefox Mobile have the highest average bug-fix times (3.3 and 4.7 months, respectively).

Bug tossing: Assigning a bug to a potential developer, also known as bug triaging, is a labor-intensive, time-consuming and fault-prone process if done manually. Moreover, bugs frequently get re-assigned to multiple developers before they are resolved, a process known as bug tossing [10]. Prior work has shown that bug-fix time increases with the increase in the tossing length of the bug, i.e., the number of times the bug has been re-assigned [10], [13]. In this section we study how often bugs are tossed in the Firefox Mobile project; note that bug toss information is not available in Google Code's bug tracker. In Fig. 2 we present the distribution of bug tossing lengths in Firefox Mobile. Similar to our prior results on studying tossing on desktop applications [13], we found that only a very low percentage of bugs (11%) are fixed by the first assigned developer, i.e., 0 tosses; for about 75% of the bugs, resolution takes 2–13 tosses, while 17% of the bugs require more than 13 tosses. We believe Google Code would benefit greatly from

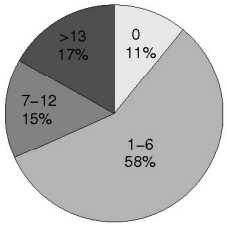


Figure 2. Distribution of number of bug tosses in Firefox Mobile.

Project	BugFixTime (months)	
	median	*average*
Platform	0	2.43
ADW Launcher	0	0.60
Notifier	0	0.31
SMSPopup	1	1.66
WifiTether	0	1.37
XBMC Remote	1	1.56
AnkiDroid	2	3.30
CallMeter3G	0	1.02
ConnectBot	1	1.58
CSipSimple	0	1.34
CyanogenMod	0	1.12
Firefox Mobile	0	4.79
GAOSP	0	0.74
IMSDroid	0	1.46
JustPictures	0	1.19
K-9	0	2.21
My Tracks	0	1.78
OpenIntents	1	1.58
OsmAnd	0	1.17
OSMDroid	1	2.08
Sipdroid	0	1.62
SoftKeyboard	0	1.9
TransDroid	0	1.2
WebSMSDroid	0	0.99
ZXing	0	0.66

Table IV
BUG-FIX TIME.

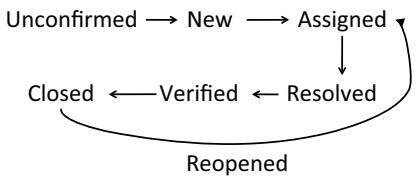
adopting automatic, machine learning-based, bug triaging and toss reduction techniques [13].

IV. ANDROID BUG LIFE-CYCLE

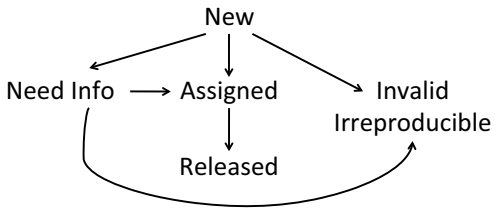
Prior work has shown that most desktop application bugs go through several stages before being finally closed [2], [10], as illustrated in Fig. 3(a). The Android-based apps we examined use the Google Code project hosting site, which has a built-in bug tracker, with the exception of Firefox Mobile, which uses Bugzilla. [7] We now turn to comparing the bug life-cycle of applications hosted on Bugzilla (Fig. 3(a)) with the bug life-cycle of Android apps hosted on Google Code (Fig. 3(b)).[8] We found two major differences between the life-cycles. First, when a bug is reported in Bugzilla, unless the bug has been investigated by a developer or person

[7]Bugzilla is used by large software projects, both open-source (Firefox, Eclipse, Apache, etc.) and commercial (Facebook, Nokia, NASA, etc.) [14].

[8]Diagram constructed from official Google Code documentation,http://source.android.com/source/life-of-a-bug.html



(a) Applications hosted on Bugzilla.



(b) Android apps hosted on Google Code.

Figure 3. Comparison of bug life-cycles.

in charge of that module, the bug status is $Unconfirmed$. On the other hand, as soon as a bug is reported in Android-based apps, the bug has the status New. This shows that in the Android projects hosted on Google Code there is no difference between a real bug (which has not been assigned or investigated yet) and an unconfirmed bug. Second, in Google Code projects there is no patch verification stage before the bug is closed and released; this is in contrast Bugzilla-based projects (e.g., Mozilla or Eclipse), where bug patches are reviewed and tested by super-reviewers in the verification stage and closed when the new bug-fix code is released. Therefore, in Android-based apps, as long as the bug is in the $Assigned$ stage it is not possible to infer the progress of the bug resolution process from bug status alone. Third, there is no status $Reopened$ for Android-based apps. For example, consider bug 4784 from the Android Platform. [9] This bug is a duplicate of Bug 3006 and the bug description of 4784 says that the bug "that was closed is still active, for the Motorola Droid running Android OS 2." This shows that bug 3006 was closed, but later re-opened after bug 4784 was filed. However, bug 4784 is not marked explicitly as $Duplicate$, and the history of bug 3006 does not indicate that it has been re-opened. As a result, developers must read through the bug report comments to track changes in bug status. Therefore, we believe that incorporating bug-tracker and change-history locating techniques from Bugzilla into Google Code would improve the bug-fixing process [2].

V. FOCUS: SECURITY BUGS

Android apps have access to a wealth of security-sensitive data such as user's location, list of contacts, microphone or camera. To better understand security issues, we performed an investigation of security bugs in the top-4 apps, ranked

[9]The report for bug BUGID in app APP is available at http://code.google.com/p/APP/issues/detail?id=BUGID.

by the number of security bug reports, among the apps we considered; we limit our analysis to only 4 apps because they had a significant number of security bugs (over 100 security bugs each). First, we show how we identify security bug reports. Second, we categorize the security bugs into different classes, e.g., licensing issues, or certification problems. Third, we perform an analysis to understand if the bug-fixing process for security bugs and the quality of security bug reports are significantly different than for non-security bugs.

A. Identifying Security Bug Reports

On Google Code, bug reports do not have specific security-based tags or labels. Therefore, we used the technique proposed by Gegick et al. [15] for identifying security bug reports. [10] We performed the following steps:

- Stop Word Removal: the first step is to remove common stop words from the bug reports.
- Identifying potential security bug reports: next, we identify bug reports which contains words like "security," "vulnerability," "attack," "crash," "buffer overflow," and "buffer overrun".
- `Tf-idf`: finally, we apply *term-frequency invert document frequency*—a common technique used in text mining to understand the significance of a word in a document and across multiple-documents [16]—to the potential security bug reports to further narrow the security bug report list.

After we have identified security bug reports in the Android-based apps, we focus on the top-4 apps, ranked by the number of security bug reports in the app. Our analysis includes 980 bugs from Android Platform, 251 bugs from CyanogenMod, 121 bugs from K-9 and 113 bugs from Firefox Mobile. We also considered the 2,357 security bugs from Mozilla's Bugzilla database which are labeled as `Component:Security`.

B. Security Bug Categorization

To understand the different categories security bugs belong to, and how categories vary across apps, we studied the frequent terms occurring in security bug reports in the four focus apps. We report the term frequency using pie-charts in Fig. 4.

We supplement this quantitative analysis with a qualitative analysis based on manual inspection of the bug reports. We found that security bugs span a variety of categories: permission issues, licensing–authorization–certification issues, injection attacks, password problems (e.g., Android Platform bug 10809), or phone locking issues (e.g., Android Platform bug 6615).

A significant number of security bugs in Android-based apps stem from events leading to permission breach; such breaches are problematic as evidence suggests that Android users are often unaware of various security and privacy issues that are associated with using their phone [17]. Therefore, we performed a closer examination of permission-based security bugs and found that they fall into two main sub-categories: permission abuse and confidentiality issues.

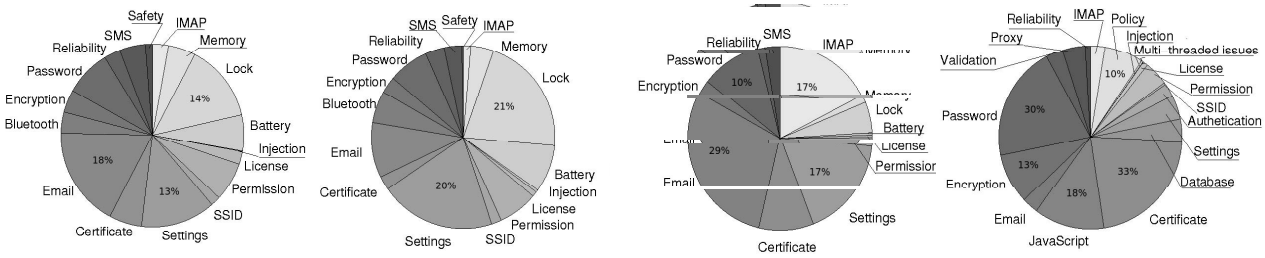
Abuse of permissions. We found that most permission-based security bugs in Android-based apps are manifested when apps abuse permissions, i.e., access data they should not have received permission to access in the first place. For example, consider Fennec bug 650509. When a user uses Fennec (i.e., Firefox for Android), the user's search or other data associated solely with the Fennec app can be used by other apps on Android. Similarly, in bug 4213 for K-9, we find that the user can be tracked via the `poster` attribute of the HTML5 `video` tag in the email app even before the user allows the images/video to be loaded. Another instance is bug 24104 from Android Platform which shows that in addition to reading sensitive data, like the geographical location of the user or the call history, apps can also *modify* phone settings even when the user thinks the app does not have the permission to so. In this case, the permission description is misleading, resulting in a false sense of security for the user; the report for bug 10412 in the Android Platform indicates that the camera app automatically adds geotags to photos taken using the phone camera, which reveals the user's location history.

Confidentiality issues. We found several bugs that lead to confidentiality breaches in the Android Platform, e.g., bugs 9392 and 18565. These bugs allow the app to accidentally send text messages to either a random contact in the user's contacts list or to someone whose number does not even exist in the contact list. Apart from breaching confidentiality, bugs like this are also hard to notice right away because the app confirms that the text message was delivered to the recipient the user intended. Another similar bug is the Android phone screen displaying the contents of a text message in the notification area even when the phone is locked (Android Platform bug 26699). This is problematic as potentially confidential messages are displayed even when the phone is locked. Another potential breach of confidentiality stems from apps not having provisions for the user to delete the data they store; for example, if the user wants to delete all data about places he has visited or movies he has rated, the Android backup app does not have provisions for the user to delete specific portions from an existing backup (bug 14581).

C. Comparing Security Bugs with Non-Security Bugs

In this section, we compare security bugs with non-security bugs in the Android-based apps using the various bug-report quality and bug-fix process based metrics introduced in Section III. We observed several trends:

[10] Note that the technique proposed by Gegick et al. [15] has some threats to validity, as discussed in Section VI.



(a) Android Platform (b) CyanogenMod (c) K-9 (d) Firefox Mobile

Figure 4. Term frequency in security bug reports.

Project	DescriptionLength (words)		ReproduceSteps (%)		OutputDetails (%)		AdditionalInfo (%)		AllDetails (%)	
	Security	*Others*	*Security*	*Others*	*Security*	*Others*	*Security*	*Others*	*Security*	*Others*
Android Platform	204.6	136.05	0.23	0.16	0.12	0.12	0.12	0.04	0.12	0.03
CyanogenMod	1135.79	307.97	80.08	74.08	0.4	0.19	6.78	10.23	0	0.02
Firefox Mobile	94.40	154.5	0.89	2.49	6.20	6.34	2.49	6.34	0	0
K-9	395.23	154.58	67.77	62.72	53.72	49.22	40.50	31.41	38.02	29.72

Table VI
BUG DESCRIPTION METRICS FOR SECURITY VS. OTHER BUGS.

Project	BugFixTime (months)			
	Security		*Others*	
	median	average	median	average
Android Platform	1	3.34	0	2.39
CyanogenMod	2	3.91	0	0.98
Firefox Mobile	0	4.89	0	4.78
K-9	1	3.55	0	2.15

Table V
BUG-FIX TIME OF SECURITY VS. OTHER BUGS.

Project	FirstComment (days)		TotalComments	
	Security	*Others*	*Security*	*Others*
Android Platform	0.82	0.68	33.01	4.66
CyanogenMod	0.05	0.04	26.78	8.02
Firefox Mobile	3.73	2.92	13.29	8.08
K-9	0.33	0.53	11.38	5.28

Table VII
COMMENT TIME AND NUMBER OF COMMENTS FOR SECURITY VS. OTHER BUGS.

Bug-fix time: In Tab. V we report the bug-fix times of the security and non-security bugs for both Google Code-based apps and the Bugzilla-based Mozilla project. In Google Code-based apps, the median time to fix a security bug is similar to the bug-fix time of non-security bugs, which indicates that security bugs are not treated with higher urgency. However, the median bug-fix time of Mozilla security bugs is much lower compared with the non-security bugs, indicating that security bugs in Mozilla are given higher urgency compared to other bugs.

Bug-Report Quality: We compare the four bug-report quality metrics, namely $DescriptionLength$, $ReproduceSteps$, $OutputDetails$, and $AdditionalInfo$ and report the results in Tab. VI. We observe that the $DescriptionLength$ of security bugs is significantly higher compared to non-security bugs for both Google Code and Bugzilla. However, for the remaining report quality metrics ($ReproduceSteps$, $OutputDetails$, and $AdditionalInfo$), we find that the values for security bugs are slightly higher compared to non-security bugs. To summarize, the higher values of bug-report quality metrics indicate that security-bug reports have higher quality compared to non-security bug reports.

Community Activity: In Tab. VII we report the values for metrics $FirstComment$ and $TotalComments$ which we defined in Section III. We observe that the time until the first comment is lower for security bugs for most apps in contrast to non-security bugs. Similarly, we find that number of $TotalComments$ is significantly higher for security bugs, compared to non-security bugs.

To conclude, we found that the quality of both bug reports and the bug-fix process is higher for security bugs compared than for non-security bugs.

VI. THREATS TO VALIDITY

We now present possible threats to the validity of our study.

External Validity. We chose 25 popular open source Android projects for our study, with projects spanning multiple categories to reduce selection bias. However, many popular apps on Google Play [4] do not have publicly-available bug reports. Hence, we cannot claim that our findings generalize to all Android software projects. We found that the Android Platform maintains a private bug

database which is not hosted on Google Code [12]. We did not consider the bug reports from the private database in our study, which might affect our results. Additionally, the security bug study (Section V) was performed on an smaller number of apps (4), which further reduce the generality of security bug conclusions.

Internal Validity. In our study we collected bug reports from Bugzilla for Firefox Mobile. For computing bug-fix time, we used the bugs marked as "closed". Our results might be affected if any of those bugs are will be reopened in the future.

Construct Validity. Construct validity relies on the assumption that our metrics actually capture the intended characteristic, e.g., bug fix time accurately models process quality, bug description metrics accurately model bug-report quality. We intentionally used multiple metrics in each of our analyses to mitigate this threat. Gegick et al.'s approach for identifying security bug reports is based on text-mining heuristics and in the original version the authors report up to 78% prediction accuracy [15]. Since we use their algorithm, we inherit potential inaccuracies in identifying security bug reports which in turn could affect our results. Some apps we considered were developed and maintained by a very small number of developers (10 or less); results from analyses of these apps have lower statistical significance levels. We used commit log messages to determine if a commit is related to a bug-fix; if the log message accidentally omits this information, we might run into the risk of missing bugs and that might affect our developer counts in Fig. 1.

Content Validity. Highly-critical security bug reports are sometimes purposefully removed from the bug databases by project managers, to reduce the risk of further aggravation of the security issue [18]. As a consequence, we run the risk of missing such security bug reports in our study.

VII. Related Work

Android bugs: Maji et al. [19] performed a failure characterization study on the Android and Symbian mobile platforms. They collected data on bugs in the OS, middleware/library, and development tools/core applications for these two platforms. Their study was focused on the relation between bugs and code, e.g., bug location (which subsystem contains the bug), fixes (what source code change was made to fix the bug), defect density, and code complexity. We also study bugs in the Android Platform, though with a different, process-oriented goal: bug life-cycle and fix-times, bug reports, etc. In addition to the platform, we collect data on 24 popular apps. As part of our own prior work on automating Android GUI testing [20] we conducted a small-scale bug counting study (10 Android apps with the time frame generally being less than an year) to identify the most prevalent bug categories, e.g., GUI bugs, unhandled exception, I/O or concurrency bugs. In contrast, the scope of the current work is broader (platform and 24 popular apps), the time frame is longer (3–4 years), and the focus is different.

Empirical bug studies: Bettenburg et al. [9] conducted an empirical study based on a survey of developers and users of Apache, Eclipse, and Mozilla to understand what makes a good bug report. Their study suggested that steps to reproduce, stack traces, and test cases are helpful information to developers. Using these results, they built a tool named Cuezilla that could measure the quality of new bug reports and recommends which elements should be added to improve the quality. Zaman et al. [21] conducted an empirical study to understand the difference between security and performance bugs in Firefox. They compared bug-fix times of security and performance bugs and found that security bugs are fixed faster than performance bugs. We performed an additional analysis on bug report quality for security vs. non-security bugs and showed that security bug reports are superior to non-security bugs. They also found that a high percentage of security bugs are re-opened in the future; lack of bug reopening information prevents us from investigating this on our examined projects.

VIII. Conclusions

In this paper we performed a set of empirical analyses to understand the bug-fixing process in the Android platform and Android-based apps. We found that, although the apps we considered were started recently, their bug reports are of high quality. We also found that the quality of security bug reports is higher compared to non-security bugs, though security bugs are fixed slower compared to other bugs. We also found that Google Code's bug tracker, which is used by most open-source Android apps, offers less bug management support—e.g., for bug triaging— compared to other widely-used trackers such as Bugzilla or Jira; this lack of information limits empirical analyses and might hinder the bug-fixing process on Google Code-based projects.

Acknowledgements

We thank Lorenzo Gomez, Steve Suh, Xuetao Wei, and the anonymous reviewers for their feedback. This work was supported in part by the National Science Foundation awards CNS-1064646 and CCF-1149632.

References

[1] Nielsen, "Smartphones Account for Half of all Mobile Phones, Dominate New Phone Purchases in the US," March 29, 2012, http://blog.nielsen.com/nielsenwire/category/online_mobile/.

[2] A. Zeller, *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2006.

[3] K. Falkner, "The big (small) survey on Android Security and Gender," April 2012, http://blog.spamfighter.com/general/the-big-small-survey-on-android-security-and-gender-infographic.html.

[4] "Google play," April 2012, https://play.google.com/.

[5] "Android platform," January 2012, http://code.google.com/p/android/.

[6] H. Dediu, "When will android reach one billion users?" February 2012, http://www.asymco.com/2012/02/29/when-will-android-reach-one-billion-users/.

[7] "Firefox for android," January 2012, http://www.mozilla.org/en-US/mobile/.

[8] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 34–43.

[9] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ser. SIGSOFT '08/FSE-16, 2008, pp. 308–318.

[10] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09, 2009, pp. 111–120.

[11] "Clarifications on Ending Contributions to Mozilla," August 2011, http://tylerdowner.wordpress.com/2011/08/27/some-clarification-and-musings/.

[12] "Bug Triaging in Android," February 2012, http://source.android.com/source/life-of-a-bug.html.

[13] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *ICSM*, 2010, pp. 1–10.

[14] "Bugzilla Users," April 2012, http://www.bugzilla.org/installation-list.

[15] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, may 2010, pp. 11 –20.

[16] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[17] "Survey of Android Users," April 2012, http://www.retrevo.com/content/blog/2011/08/iphones-backups-and-toilets-connection.

[18] "Android security bugs wiki," January 2012, https://developer.android.com/resources/faq/security.html.

[19] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, nov. 2010, pp. 249 –258.

[20] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11, 2011, pp. 77–83.

[21] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 93–102.