A Cross-platform Analysis of Bugs and Bug-fixing in Open Source Projects: Desktop vs. Android vs. iOS

Bo Zhou Iulian Neamtiu Rajiv Gupta
Department of Computer Science and Engineering
University of California Riverside, CA, USA
{bzhou003, neamtiu, gupta}@cs.ucr.edu

ABSTRACT

As smartphones continue to increase in popularity, understanding how software processes associated with the smartphone platform differ from the traditional desktop platform is critical for improving user experience and facilitating software development and maintenance. In this paper we focus specifically on differences in bugs and bug-fixing processes between desktop and smartphone software. Our study covers 444,129 bug reports in 88 open source projects on desktop, Android, and iOS. The study has two main thrusts: a quantitative analysis to discover similarities and differences between desktop and smartphone bug reports/processes; and a qualitative analysis where we extract topics from bug reports to understand bugs' nature, categories, and differences between platforms. Our findings include: during 2011–2013, iOS bugs were fixed three times faster compared to Android and desktop; top smartphone bug fixers are more involved in reporting bugs than top desktop bug fixers; and most frequent high-severity bugs are due to build issues on desktop, concurrency on Android, and application logic on iOS. Our study, findings, and recommendations are potentially useful to smartphone researchers and practitioners.

1. INTRODUCTION

Smartphones and the applications ("apps") running on them continue to grow in popularity [26] and revenue [16]. This increase is shifting client-side software development and use, away from traditional desktop programs and towards smartphone apps [13, 15].

Smartphone apps are different from desktop programs on a number of levels: novelty of the platform (the leading platforms, Android and iOS, have become available in 2007), construction (sensor-, gesture-, and event-driven [9]), concerns (security and privacy due to access to sensitive data), and constraints (low memory and power consumption).

Empirical bugs and bug-fixing studies so far have mostly focused on traditional software; few efforts [5, 17] have investigated the differences between desktop and smartphone software. Therefore, in this paper we analyzed the similarities and differences in bug reports and bug-fixing processes between desktop and smartphone platforms. Our study covers 88 projects (34 on desktop, 38 on Android, 16 on iOS) encompassing 444,129 bug reports. We analyzed bugs in a time span beginning in 1998 for desktop and 2007 for Android/iOS, and ending at the end of December 2013.

In particular, we studied the bug-fix process features, bug nature and the reporter/fixer relationship to understand how bugs, as well as bug-fixing processes, differ between desktop and smartphone. Section 2 describes our methodology, including how we selected projects, the steps and metrics we used for extracting bug reports, process features, and topics.

The study has two thrusts. First, a *quantitative* thrust (Section 3) where we compare the three platforms in terms of attributes associated with bug reports and the bug-fixing process, how developer profiles differ between desktop and smartphone, etc. Second, a *qualitative* thrust (Section 4) where we apply LDA to extract topics from bug reports on each platform and gain insights into the nature of bugs, how bug categories differ from desktop to smartphone, and how these categories change over time.

We now present some highlights of our findings:

- Bug-fixing process features (e.g., fix time, comments) differ between desktop and the two smartphone platforms, but are similar for Android and iOS.
- The most important issues differ across platforms: the most frequent high-severity bugs are due to compilation and validation failures on desktop (56%), whereas on Android they are due to concurrency (66%), and on iOS due to application crashes (52%).
- Concurrency bugs are much more prevalent on Android than on iOS.
- Despite the attention they have received in the research community, we found that issues commonly associated with smartphone apps such as energy, security and performance, are not very prevalent.

In light of our findings, in Section 5 we provide a set of recommendations for improvement.

Reproducibility: the complete datasets used in our analyses, as well as supplementary materials, are available online at `http://www.cs.ucr.edu/~bzhou003/cross_platform.html`.

2. METHODOLOGY

We first provide an overview of the examined projects, and then describe how we extracted bug features and topics.

2.1 Examined Projects

We chose 88 open source projects for our study, spread across the three platforms: 34 desktop projects, 38 Android

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EASE '15, April 27 - 29, Nanjing, China
Copyright 2015 ACM 978-1-4503-3350-4/15/04 ...$15.00.

Table 1: Projects examined, bugs reported, bugs fixed, and time span.

Desktop

Project	Bugs Reported	Bugs Fixed (FixRate)	Time span
Mozilla Core	247,376	101,647 (41.09%)	2/98-12/13
OpenOffice	124,373	48,067 (38.65%)	10/00-12/13
Gnome Core	160,825	42,867 (26.65%)	10/01-12/13
Eclipse platform	100,559	42,401 (42.17%)	2/99-12/13
Eclipse JDT	50,370	22,775 (45.22%)	10/01-12/13
Firefox	132,917	19,312 (14.53%)	4/98-12/13
SeaMonkey	91,656	18,831 (20.55%)	4/01-12/13
Konqueror	38,156	15,990 (41.91%)	4/00-12/13
Eclipse CDT	17,646	10,168 (57.62%)	1/02-12/13
WordPress	26,632	9,995 (37.53%)	6/04-12/13
KMail	21,636	8,324 (38.47%)	11/02-12/13
Linux Kernel	22,671	7,535 (33.24%)	3/99-12/13
Thunderbird	39,323	5,684 (14.45%)	4/00-12/13
Amarok	18,212	5,400 (29.65%)	11/03-12/13
Plasma Desktop	22,187	5,294 (23.86%)	7/02-12/13
Mylyn	8,461	5,050 (59.69%)	10/05-12/13
Spring	15,300	4,937 (32.27%)	8/00-12/13
Tomcat	11,332	4,826 (42.59%)	11/03-12/13
MantisBT	11,484	4,141 (36.06%)	2/01-12/13
Hadoop	11,444	4,077 (35.63%)	10/05-12/13
VLC	9,674	3,892 (40.24%)	5/05-12/13
Kdevelop	7,824	3,572 (45.65%)	8/99-12/13
Kate	7,058	3,326 (47.12%)	1/00-12/13
Lucene	5,327	3,035 (56.97%)	4/02-12/13
Kopete	9,824	2,957 (30.10%)	10/01-9/13
Hibernate	8,366	2,737 (32.72%)	10/00-12/13
Ant	5,848	2,612 (44.66%)	4/03-12/13
Apache Cassandra	3,609	2,463 (68.25%)	8/04-12/13
digikam	6,107	2,400 (39.30%)	3/02-12/13
Apache httpd	7,666	2,334 (30.45%)	2/03-10/13
Dolphin	7,097	2,161 (30.45%)	6/02-12/13
K3b	4,009	1,380 (34.42%)	4/04-11/13
Apache Maven	2,586	1,332 (51.51%)	10/01-12/13
Portable OpenSSH	2,206	1,061 (48.10%)	3/09-12/13
Total	*1,259,758*	*422,583 (33.54%)*	

Android

Project	Bugs Reported	Bugs Fixed (FixRate)	Time span
Android Platform	64,158	3,497 (5.45%)	11/07-12/13
Firefox for Android	11,998	4,489 (37.41%)	9/08-12/13
K-9 Mail	6,079	1,200 (19.74%)	6/10-12/13
Chrome for Android	3,787	1,601 (42.28%)	10/08-12/13
OsmAnd Maps	2,253	1,018 (45.18%)	1/12-12/13
AnkiDroid Flashcards	1,940	746 (38.45%)	7/09-12/13
CSipSimple	2,584	604 (23.37%)	4/10-12/13
My Tracks	1,433	525 (36.64%)	5/10-12/13
CyanogenMod	788	432 (54.82%)	9/10-1/13
Andro-minion	623	346 (55.54%)	9/11-11/13
WordPress for Android	532	317 (59.59%)	9/09-9/13
Sipdroid	1,149	300 (26.11%)	4/09-4/13
AnySoftKeyboard	1,144	229 (20.02%)	5/09-5/12
libphonenumber	389	219 (56.30%)	11/07-12/13
ZXing	1,696	218 (12.85%)	5/09-12/12
SL4A	701	204 (29.10%)	10/09-5/12
WebSMSDroid	815	197 (24.17%)	7/10-12/13
OpenIntents	553	188 (34.00%)	12/07-6/12
IMSDroid	502	183 (36.45%)	6/10-3/13
Wikimedia Mobile	261	166 (63.60%)	1/09-9/12
OSMdroid	494	166 (33.60%)	2/09-12/13
WebKit	225	157 (69.78%)	11/09-3/13
XBMC Remote	729	129 (17.70%)	9/09-11/11
Mapsforge	466	127 (27.25%)	2/09-12/13
libgdx	384	126 (32.81%)	5/10-12/13
WiFi Tether	1,938	125 (6.45%)	11/09-7/13
Call Meter NG/3G	904	116 (12.83%)	2/10-11/13
GAOSP	529	114 (21.55%)	2/09-5/11
Open GPS Tracker	391	114 (29.16%)	7/11-9/12
CM7 Atrix	337	103 (30.56%)	3/11-5/12
Transdroid	481	103 (21.41%)	4/09-10/13
MiniCM	759	101 (13.31%)	4/10-5/12
Connectbot	676	87 (12.87%)	4/08-6/12
Synodroid	214	86 (40.19%)	4/10-1/13
Shuffle	325	77 (36.56%)	10/08-7/12
Eyes-Free	322	69 (21.43%)	6/09-12/13
Omnidroid	184	61 (33.15%)	10/09-8/10
VLC for Android	151	39 (25.83%)	5/12-12/13
Total	*112,894*	*18,579 (27.28%)*	

iOS

Project	Bugs Reported	Bugs Fixed (FixRate)	Time span
WordPress for iPhone	1,647	892 (54.16%)	7/08-9/13
Cocos2d for iPhone	1,506	628 (41.70%)	7/08-5/13
Core Plot	614	218 (35.50%)	2/09-12/13
Siphon	586	162 (27.65%)	4/08-11/11
Colloquy	542	149 (27.49%)	12/08-12/13
Chrome for iOS	365	129 (35.34%)	6/09-12/13
tweetero	142	109 (76.76%)	8/09-6/10
BTstack	360	106 (29.44%)	2/08-12/13
Mobile Terminal	311	82 (26.37%)	8/07-3/12
MyTime	247	101 (40.89%)	7/11-11/13
VLC for iOS	188	80 (42.55%)	8/07-12/13
Frotz	214	78 (36.45%)	9/10-9/12
iDoubs	164	74 (45.12%)	9/07-7/13
Vnsea	173	58 (33.53%)	4/08-10/10
Metasyntactic	145	50 (34.48%)	7/08-4/12
Tomes	148	51 (34.46%)	8/07-5/08
Total	*7,352*	*2,967 (37.40%)*	

projects, and 16 iOS projects. We used several criteria for choosing these projects and reducing confounding factors. First, the projects we selected had large user bases, e.g., on desktop we chose [1] Firefox, Eclipse, Apache, KDE, Linux kernel, WordPress, etc.; on Android, we chose Firefox for Android, Chrome for Android, Android platform, K-9 Mail, WordPress for Android; on iOS we chose Chrome for iOS, VLC for iOS, WordPress for iOS, etc. Second, we chose projects that are popular, as indicated by the number of downloads and ratings on app marketplaces. For the Android projects, the mean number of downloads, per Google Play, was 1 million, while the mean number of user ratings was 7,807. For the iOS projects, the mean number of ratings on Apple's App Store was 3,596; the store does not provide the number of downloads. Third, we chose projects that have had a relatively long evolution history ("relatively long" because the Android and iOS platforms emerged in 2007). Fourth, to reduce selection bias, we choose projects from a wide range of categories—browsers, media players, utilities, infrastructure.

Table 1 shows a summary of the projects we examined. For each platform, we show the project's name, the number of reported bugs, the number of closed and fixed bugs, the FixRate (i.e., the percentage of fixed bugs in the total number of reported bugs), and finally, the dates of the first and last bugs we considered.

2.2 Collecting Bug Reports

We now describe the process used to collect data. All 88 projects offer public access to their bug tracking systems. The projects used various bug trackers: desktop projects tend to use Bugzilla, Trac, or JIRA, while smartphone projects use mostly Google Code, though some use Bugzilla or Trac. We used Scrapy,[2] an open source web scraping tool, to crawl and extract bug report features from bug reports located in each bug tracking system.

[1] Many of the desktop projects we chose have previously been used in empirical studies [3, 4, 8, 12, 18, 34].

[2] http://scrapy.org

For bug repositories based on Bugzilla, Trac, and JIRA, we only considered bugs with resolution `RESOLVED` or `FIXED`, and status `CLOSED`, as these are confirmed bugs; we did not consider bugs with other statuses, e.g., `UNCONFIRMED` and other resolutions, e.g., `WONTFIX`, `INVALID`. For Google Code repositories, we selected bug reports with type `defect` and status `fixed`, `done`, `released`, or `verified`.

2.3 Quantitative Analysis

To find quantitative differences in bug-fixing processes we performed an analysis on various features (attributes) of the bug-fixing process, e.g., fix time, severity, comment length. We now provide definitions for these features.

FixTime: the time required to fix the bug, in days, computed from the day the bug was reported to the day the bug was closed. *Severity* is an indicator of the bug's potential impact on customers. Since severity levels differ among trackers, we mapped severity from different trackers to a uniform 10-point scale, as follows: 2=Trivial/Tweak, 5=Minor/Low/Small, 6=Normal/Medium, 8=Major/Crash/High, 9=Critical, 10=Blocker. *BugTitle:* the text content of the bug report title. *BugDescription:* the text content of the bug summary/description. *DescriptionLength:* the number of words in the bug summary/description. *TotalComments:* the number of comments in the bug report. *CommentLength:* the number of words in all the comments attached to the bug report.

Data preprocessing: feature values and trends. We computed per-project values at monthly granularity, for several reasons: (1) to also study differences between projects within a platform; (2) to avoid data bias resulting from over-representation, e.g., Mozilla Core bugs account for 24% of total desktop bugs, hence conflating all the bug reports into a single "desktop" category would give undue bias to Mozilla; and (3) we found monthly to be a good granularity for studying trends. For each feature, e.g., FixTime, we compute the mean [3] and the trend (slope) as follows:

Input: Feature value per bug
 for each project **do**
  for i = start month **to** last month **do**
   feature[i] = geometric.mean(input)
  end for
  FeatureMean = geometric.mean(feature)
  FeatureBeta = slope(feature $\sim$ time)
 end for
Output: FeatureMean, FeatureBeta

We employed three statistical tests in our analysis:

Trend test. To test whether a feature increases/decreases over time, we build a linear regression model where the independent variable is the time and the dependent variable is the feature value for each project. We consider that the trend is increasing (or decreasing, respectively) if the slope β of the regression model is positive (or negative, respectively) and $p < 0.05$.

Non-zero test. To test whether a set of values differs significantly from 0, we perform a one-sample t-test where the specified value was 0; if $p < 0.05$, we consider that the samples differ from 0 significantly.

Pairwise comparison test. To check whether feature values differ significantly between platforms, we conducted pairwise comparisons (desktop v. Android; desktop v. iOS; and Android v. iOS) using the Wilcoxon-Mann-Whitney test.

2.4 Qualitative Analysis

For the second thrust of our paper, we used a qualitative analysis to understand the nature of the bugs by extracting topics from bug reports. We used the bug title, bug description and comments for topic extraction. We applied several standard text retrieval and processing techniques for making text corpora amenable to text analyses [30] before applying LDA: stemming, stop-word removal, non-alphabetic word removal, programming language keyword removal. We then used MALLET [22] for topic training. The parameter settings are presented in Section 4.1.

3. QUANTITATIVE ANALYSIS

The first thrust of our study takes a quantitative approach to investigating the similarities and differences between bug-fixing processes on desktop and smartphone platforms. Specifically, we are interested in how bug-fixing process attributes differ across platforms; how the contributor sets (bug reporters and bug owners) vary between platforms; how the bug-fix rate varies and what factors influence it.

3.1 Bug-fix Process Attributes

We start with the quantitative analysis of bug characteristics and bug-fixing process features. We show the results, as beanplots, in Figures 1 through 5. The shape of the beanplot is the entire density distribution, the short horizontal lines represent each data point, the longer thick lines are the medians, and the white diamond points are the geometric means. We now discuss each feature.

FixTime. Several observations emerge. First, *desktop bugs took longer to fix than smartphone bugs: 99 days on desktop, 28 days on Android, 24 days on iOS* (Figure 1a). The pairwise comparison test indicates that FixTime on desktop differs from Android and iOS ($p \ll 0.01$ for both); there is no statistical difference between Android and iOS ($p = 0.8$). This is due to multiple reasons, mainly low severity and large number of comments. According to previous research [4,12], FixTime is correlated with many factors, e.g., positively with number of comments or bug reports with attachments, and negatively with bug severity. As can be seen in Figure 1d, the number of comments for desktop is larger. The severity of desktop bugs is lower, as shown in Figure 1b. We have also observed (as have Lamkanfi and Demeyer [18]) that on desktop many bugs are reported in the wrong component of the software system, which prolongs fixing.

Second, *bug-fix time tends to decrease over time on desktop and iOS.* In fact, FixTime is the only feature where the non-zero test for β's turned out significant or suggestive for all platforms ($p < 0.01$ for desktop, $p = 0.124$ for Android, $p = 0.095$ for iOS). As Figure 1f shows, most desktop projects (29 out of 34) and iOS projects (11 out of 16) have decreasing trends, i.e., negative β's, on FixTime. For Android, only half of the projects (19 out of 38) have the same trends. The reasons are again multiple.

The first reason is increasing developer experience: as developers become more experienced, they take less time to fix bugs. The second reason is increased developer engagement. High overlap of bug reporters and bug owners results in shorter bug fixing time, since project developers are more familiar with their own products.

Figure 2 shows the percentage of owners who have also

[3] Since the distributions are skewed, we used the geometric instead of arithmetic mean.

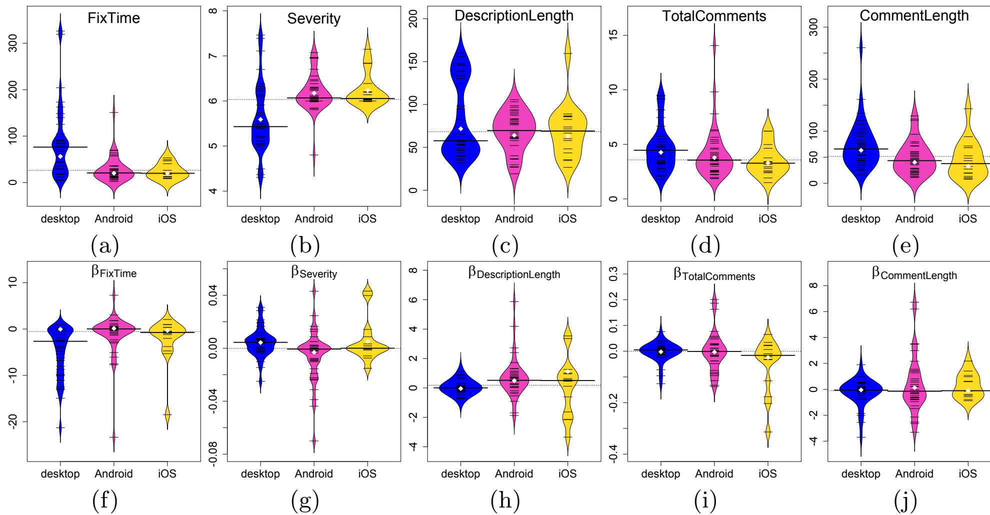

Figure 1: Beanplot of feature distributions (Figure 1a–1e) and corresponding trends (Figure 1f–1j) per project.

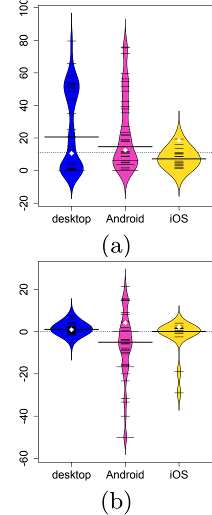

Figure 2: Percentage of bug owners who have reported bugs (a) and their trends (b).

reported at least one bug for each project and their corresponding trend—the graph reveals higher engagement over time for desktop and iOS, but not for Android (for Android, 23 out of 38 projects show lower engagement over time).

Other researchers had similar findings: Giger et al. [8] found that older bugs (e.g., Mozilla bugs opened before 2002 or Gnome bugs opened before 2005) were likely to take more time to fix than recently-reported bugs; and more recent bugs were fixed faster because of the increasing involvement of external developers and the maturation of the project [23].

Severity. High-severity bug reports indicate those issues that the community considers to be of utmost priority on each platform. Figure 1b shows that desktop bug severity is lower than smartphone bug severity. When looking at severity trends, as Figure 1g indicates, severity is steady at level 6 (Normal/Medium) for Android and iOS and has a small increasing trend for desktop (22 out of 34 projects on desktop have increasing trend). The pairwise comparison indicates severity on desktop differs from Android and iOS ($p \ll 0.01$ for both), and no statistical difference between Android and iOS ($p = 0.769$). Upon investigation, we found that in desktop, over time, the frequency of high-severity bugs (e.g., crashes or compilation issues) increases, which raises the mean severity level. We examined projects' release frequency, and saw an increasing frequency for desktop, meaning for desktop there is less time for validating new releases and a higher incidence of severe bugs. We investigate the nature of high-severity bugs in Section 4.3.

DescriptionLength. The number of words in the bug description reflects the level of detail in which bugs are described. A higher DescriptionLength value indicates a higher bug report quality [4], i.e., bug fixers can understand and find the correct fix strategy easier. The pairwise test indicates there is no statistical significant difference in DescriptionLength among platforms ($p > 0.659$ for all three cases). DescriptionLength stays constant on desktop and iOS (Figure 1h), but on Android increased significantly ($p = 0.003$). We found that the increase on Android is due to more stringent reporting requirements (e.g., asking reporters to provide steps-to-reproduce [1]).

TotalComments. Bugs that are controversial or difficult to fix have a higher number of comments. The number of comments can also reflect the amount of communication between application users and developers—the higher the number of people interested in a bug report, the more likely it is to be fixed [10]. The means differ (4.6 for desktop, 4.14 for Android, 3.5 for iOS, as shown in Figure 1d) but not significantly (all $p > 0.07$); TotalComments also tends to stay constant on all three platforms (non-zero test $p > 0.46$ in each case). For iOS, TotalComments starts lower and stays lower than for desktop and Android; we found that this is due to a smaller number of reporters and owners (which reduces the amount of communication), as well as overlap between reporters and owners (Figure 2), which reduces the need for commenting; we will provide an example shortly, from the Colloquy project.

CommentLength. This measure, shown in Figures 1e and 1j, bears some similarity with TotalComments, in that it reflects the complexity of the bug and activity of contributor community. Results were similar to TotalComments'. However, iOS has smaller CommentLength values (33) than desktop (63) and Android (40). The pairwise tests show that desktop differs with Android and iOS ($p = 0.005$ and 0.01, respectively), but there is no statistical difference between Android and iOS ($p = 0.48$). Upon examining iOS bug reports we found that fewer users are involved in iOS apps' bug-fixing—bug fixers frequently locate the bug by themselves and close the report, with little or no commenting. For instance, the mean CommentLength in the Colloquy project is just 9.63 words. Even for high-severity bugs such as Colloquy bug #3442 (an app crash, with severity Blocker) there is no communication between the bug reporter and bug owner—rather, the developer has just fixed the bug and closed the bug report.

Generality. We also performed a smaller-scale study where we control for process, and to a smaller extent developers, by using cross-platform projects. The study, which will be presented in Section 3.4, has yielded findings similar to the aforementioned ones.

3.2 Management of Bug-fixing

Resource allocation and management of the bug-fixing process have a significant impact on software development [34]; for example, traditional software quality is affected by the relation between bug reporters and bug owners [3]. We de-

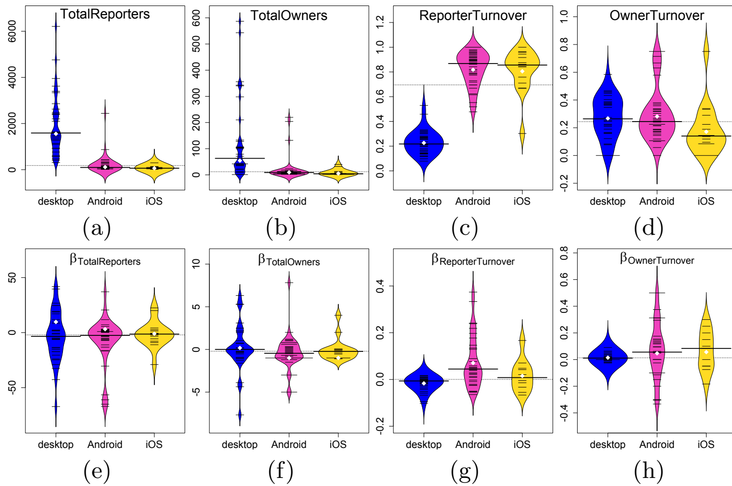

Figure 3: Owners, reporters, their turnover and trends.

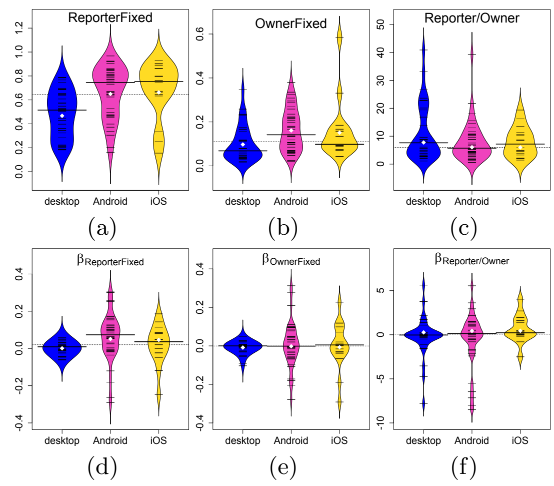

Figure 4: Beanplot of fixed developer metrics.

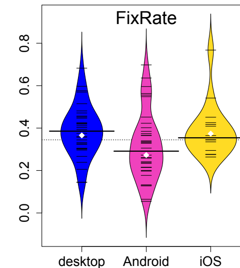

Figure 5: Bug fix rate.

fined the two roles in Section 2.3 and now set out to analyze the relationship between bug reporters and bug owners across the different platforms.

We examined the distribution and evolution of bug reporters, as well as bug owners, for the three platforms. To investigate how reporters (or owners) change overtime, we introduce a new metric, *Turnover*, i.e., the percentage of bug reporters (or owners) changed compared to the previous year. In Figures 3a, 3b, 3e and 3f we plot the numbers of bug reporters and owners for each project; we will discuss the evolution of the numbers of reporters and owners shortly. Figures 3c, 3d, 3g and 3h show the turnover per project for each platform. We make several observations.

First, desktop projects have larger sets of bug reporters and bug owners. Desktop projects also have a more hierarchical structure with front accounts for filing and fixing bugs (e.g., “issues@www” in OpenOffice for reporters, “Konqueror Developers”, “Tomcat Developers Mailing List” for owners).

Second, owner turnover is lower than reporter turnover, echoing one of our findings on bug reporting ramping up and down faster than bug owning (end of Section 3.2). The turnover of bug reporters differs significantly between desktop and smartphone ($p \ll 0.01$ for both), but not between Android and iOS ($p = 0.917$). Furthermore, the turnover of bug owners differs between desktop and iOS ($p = 0.015$) as well as Android and iOS ($p = 0.018$); the difference is not significant between desktop and Android ($p = 0.644$).

The number of fixed bugs differs across platforms, so to be able to compare reporter and owner activity between platforms, we use the number of bug reporters and bug owners in each month divided by the number of fixed bugs in that month (which we name ReporterFixed, OwnerFixed and Reporter/Owner, respectively). Figures 4 shows the result.

According to Figures 4a and 4d, ReporterFixed values for Android and iOS are higher than for desktop, which we believe is due to two reasons: higher user base and popularity of smartphone apps, and a lower effort/barrier for reporting bugs (e.g., no need to provide steps-to-reproduce as required on desktop [1]). Pairwise test results show significant differences between Android and desktop/iOS ($p \ll 0.01$ for both), but not between desktop and iOS ($p = 0.715$).

OwnerFixed is lower on desktop (Figure 4b); this measures the inverse of workload and effort associated with bug-fixing (high ratio = low workload); given the low OwnerTurnover rates for all platforms, it is unsurprising that OwnerFixed (workload) tends to stay constant for all platforms (Figure 4e). The pairwise test shows that desktop differs from smartphone platforms ($p < 0.01$) but the difference is not significant between Android and iOS ($p = 0.323$).

The ratio of reporters to owners (Figures 4c and 4f) changes in an interesting way on all platforms—increase, then decrease—which is due to users adopting applications (and finding/reporting bugs) at a faster pace than the development team is growing, hence the initial increase; eventually, as applications mature, their reporter base decreases at a faster pace than their owner base. There are no significant differences between platforms ($p > 0.19$ in all cases).

3.3 Bug Fix Rate Comparison

The bug fix rate is an indication of the efficiency of the bug-fixing process: a low fix rate can be due to many spurious bug reports being filed, or when reports are legitimate, developers are unable to cope with the high workload required for addressing the issues. Figure 5 shows the fix rate.

The mean bug fix rate for Android (27.28%) is lower than for desktop (36.56%) and iOS (37.40%). Our investigation has revealed that this is due to differences in bug reporter profiles and developer workloads.

In Android, two projects have much lower fix rates than others: Android platform (5.45%) and Wifi Tether (6.45%). When examining their workload compared to other projects, we found it to be very high (Android platform has 2,433 bug reporters and 130 bug owners, while Wifi Tether has 117 reporters but only 3 owners), which results in a low fix rate. On the other hand, WebKit, the project with the highest fix rate (69.78%), has 29 bug reporters and 18 bug owners—the high fix rate is unsurprising, given the lighter workload.

For desktop, the fix rate for Firefox (14.53%) and Thunderbird (14.45%) are the lowest. In contrast, Cassandra (68.25%),

Table 2: Mean feature values for cross-platform projects.

Project		FixTime	Severity	Description Length	Total Comments	Comment Length	Fix Rate	Reporter Turnover	Owner Turnover	Reporter Fixed	Owner Fixed	Reporter Owner
Chrome	Android	20.82	5.82	57.69	7.95	61.12	42.28%	0.61	0.22	0.40	0.26	1.50
	iOS	14.11	5.94	50.61	6.05	41.73	35.34%	0.30	0.10	0.79	0.58	1.44
Firefox	Desktop	86.20	5.61	45.19	8.59	86.30	14.53%	0.28	0.45	0.36	0.16	2.56
	Android	28.29	6.16	40.50	8.64	68.41	37.41%	0.52	0.24	0.30	0.21	1.79
WordPress	Desktop	9.54	5.79	38.37	3.38	42.01	37.53%	0.27	0.46	0.34	0.04	8.82
	Android	9.70	7.22	22.12	1.87	12.20	59.59%	0.48	0.17	0.16	0.12	2.17
	iOS	6.03	6.97	26.34	2.84	27.12	54.16%	0.76	0.34	0.25	0.13	2.04
VLC	Desktop	23.20	6.21	36.77	2.48	15.76	40.24%	0.21	0.39	0.40	0.08	5.00
	Android	18.40	6.27	22.02	2.82	11.52	25.83%	1.00	0.00	0.80	0.22	3.76
	iOS	8.96	6.77	22.30	2.24	12.02	42.55%	0.67	0.00	0.33	0.09	3.73

Mylyn (59.69%) and Eclipse CDT (57.62%) have much higher fix rates. The high rate of duplicate bug reports (27.18% for Firefox and 32.02% for Thunderbird) certainly plays a role in the low fix rate. Note, however, that Firefox and Thunderbird, a Web browser and email client respectively, are used by broad categories of people that have varying levels of expertise. In contrast, Mylyn is a task management system, Eclipse CDT is an IDE, Cassandra is a distributed database management system; their users have higher levels of expertise. Hence we believe that users of the latter applications are more adept at filing and fixing bugs than Firefox and Thunderbird users, leading to a higher fix rate.

For iOS, no application stands out as having a much lower fix rate than others. While Chrome for iOS has a low fix rate (35.34%), it is comparable with Chrome for Android (42.28%); tweetero has the highest fix rate (76.76%), understandably so as the project has 14 bug reporters and 5 bug owners.

Pairwise tests for fix rates show that the rates for desktop and Android projects differ ($p = 0.010$), as do Android and iOS projects ($p = 0.039$); the difference in fix rate between desktop and iOS projects is not significant ($p = 0.673$).

3.4 Case Study: Cross-platform Projects

We now present a method and case study for comparing process features in a more controlled setting, using cross-platform projects. We chose four apps, Chrome, Firefox, WordPress and VLC: the first two are dual-platform, while the last two are present on all three platforms. This comparison method is somewhat orthogonal to our approach so far: on one hand, it compares desktop, Android and iOS while eliminating some confounding factors, as within each project, processes and some developers[4] are common across platforms; on the other hand it uses a small set of projects. Table 2 shows the geometric mean of features and bug-fixing management metrics for each app on different platforms; differences between the means were significant ($p < 0.01$), with few exceptions.[5]

We make two observations. First, several findings (e.g., iOS bugs are fixed faster; Android bugs have larger ReporterTurnover; OwnerTurnover and CommentLength are higher on desktop) are consistent with findings in Section 3.1, which gives us increased confidence in the generality of those results. Second, researchers and practitioners can use these findings to start exploring why, within a project, the subproject associated with a certain platform fares better than the other platforms.

4. QUALITATIVE ANALYSIS

We now turn to a *qualitative analysis* that investigates the nature of the bugs. Specifically, we are interested in what kinds of bugs affect each platform, what are the most important issues (high-severity bugs) on each platform, and how the nature of bugs changes as projects evolve.

We use topic analysis; we first extract topics (sets of related keywords) via LDA from the terms (keywords) used in bug title, descriptions and comments, as described in Section 2.4 used each year in each platform, and then compare the topics to figure out how topics change over time in each platform, how topics differ across platforms, and what were the prevalent bug topics in smartphone projects.

4.1 Topic Extraction

The number of bug reports varies across projects, as seen in Table 1. Moreover, some projects are related in that they depend on a common set of libraries, for instance SeaMonkey, Firefox and Thunderbird use functionality from libraries in Mozilla Core, e.g., handling Web content. It is possible that a bug in Mozilla Core cascades and actually manifests as a crash or issue in SeaMonkey, Firefox, or Thunderbird, which leads to three separate bugs being filed in the latter three projects. For example, Mozilla Core bug #269568 cascaded into another two bugs in Firefox and Thunderbird.

Hence we extract topics using a *sampling* strategy, to reduce possible over-representation due to large projects and shared dependences. [6] More concretely, we extracted topics

[4]For Chrome, there are 337 bug reporters and 218 bug owners for Android, while the iOS version has 62 bug reporters and 38 bug owners. We found that 16 bug reporters and 13 bug owners contribute to both platforms; in fact, 6 of them reported and fixed bugs on both Android and iOS. For Firefox, we found 3,380 and 423 bug reporters for desktop and Android, respectively; 216 of them reported bugs on both platforms. We also found that Firefox has 911 and 203 bug owners on desktop and Android, respectively, with 80 owning bugs on both platforms. Finally, there were 58 developers that have reported and owned bugs on both platforms. In charge of WordPress bugs, there were 2,352, 37 and 99 bug reporters (in desktop, Android and iOS, respectively) and 205, 8 and 31 bug owners (in desktop, Android and iOS). We found that 3 reporters open bug reports on all three platforms. For bug owners, we did not find developers who contribute to both desktop and Android; though 4 developers fixed bugs in both Android and iOS, while 3 developers fixed bugs for desktop and iOS. For VLC, there were 1,451, 28 and 27 bug reporters and 98, 4 and 5 bug owners in desktop, Android and iOS, respectively; only one developer has contributed to all the platforms as bug reporter and owner.

[5]We again ran a Wilcoxon-Mann-Whitney test between feature sets on different platforms but within the same project; non-significant features were Severity and DescriptionLength for Chrome; TotalComment for Firefox; FixTime on desktop v. Android, Severity ($p = 0.873$) and DescriptionLength ($p = 0.069$) on Android v. iOS. for WordPress; and Android v. iOS on VLC.

[6]We performed a similar analysis using the original data sets in their entirety, with no sampling. As expected, the topic analysis results were influenced by the large projects, e.g., "Qt", the shared library used in KDE, was the strongest topic in 2008. We omit presenting the results on the original

Table 3: Top-5 topics in each platform per year for the sampled data set.

Year	Top 5 topics (topic weight)				
Desktop					
1999	layout (30%)	reassign (20%)	application logic (15%)	crash (14%)	php (8%)
2000	database (22%)	reassign (20%)	crash (15%)	layout (15%)	application logic (9%)
2001	crash (16%)	compilation (16%)	reassign (16%)	application logic (14%)	php (13%)
2002	application logic (14%)	compilation (14%)	crash (12%)	SCSI (12%)	ssh (12%)
2003	compilation (21%)	application logic (15%)	crash (11%)	config (9%)	connection (9%)
2004	UI (33%)	application logic (13%)	crash (12%)	config (9%)	compilation (7%)
2005	sql (28%)	application logic (15%)	crash (12%)	compilation (8%)	php (7%)
2006	application logic (17%)	Spring (16%)	crash (15%)	sql (9%)	php (7%)
2007	crash (14%)	application logic (13%)	graphic (13%)	php (10%)	connection (9%)
2008	application logic (16%)	crash (15%)	component mgmt. (12%)	UI (10%)	php (9%)
2009	crash (17%)	application logic (14%)	thread (13%)	UI (11%)	connection (7%)
2010	crash (17%)	application logic (16%)	audio (11%)	php (10%)	video (8%)
2011	debug symbol (20%)	crash (19%)	application logic (14%)	video (7%)	php (7%)
2012	crash (21%)	application logic (14%)	Hadoop (12%)	video (11%)	connection (9%)
2013	crash (21%)	Hadoop (21%)	application logic (17%)	video (8%)	connection (7%)
Android					
2008	intent (26%)	sensor (22%)	UI (21%)	thread handler (14%)	phone call (8%)
2009	UI (17%)	phone call (15%)	thread handler (12%)	intent (10%)	wifi (10%)
2010	phone call (18%)	UI (16%)	thread handler (13%)	battery (9%)	wifi (8%)
2011	thread handler (19%)	UI (14%)	network (12%)	phone call (10%)	reboot (9%)
2012	thread handler (18%)	map (16%)	UI (14%)	phone call (10%)	locale (9%)
2013	UI (21%)	scale (17%)	API (12%)	phone call (11%)	thread handler (11%)
iOS					
2007	ebook (29%)	screen display (29%)	general (15%)	UI (13%)	compilation (4%)
2008	general (30%)	multimedia (27%)	screen display (19%)	compilation (4%)	MyTime (3%)
2009	Siphon (28%)	general (26%)	message (15%)	screen display (9%)	compilation (5%)
2010	multimedia (22%)	general (19%)	graph plot (15%)	screen display (15%)	compilation (10%)
2011	SDK (44%)	general (19%)	compilation (8%)	screen display (8%)	graph plot (8%)
2012	BTstack (30%)	general (21%)	graph plot (16%)	UI (10%)	screen display (9%)
2013	sync (39%)	compilation (20%)	general (16%)	UI (7%)	WordPress (6%)

from 1,000 "independent" bug reports for each project group, e.g., Mozilla, KDE. The independent bug report sets were constructed as follows: since we have 10 projects from KDE, we sampled 100 bugs from each KDE-related project. We followed a similar process for Mozilla, Eclipse and Apache. Android and iOS had smaller number of bug reports, so for Android we sampled 100 bug reports from each project, and for iOS we sampled 50 bug reports from each project.

We used LDA (as described in Section 2.4) on the sampled sets; since there were only 2 bug reports on 1998 for desktop and 1 for Android in 2007, we have omitted those years. The preprocessing of desktop, Android, and iOS sets resulted in 824,275 words (37,891 distinct), 238,027 words (12,046 distinct) and 71,869 words (5,852 distinct), respectively. In the next step, we used MALLET [22] for LDA computation. We ran for 10,000 sampling iterations, the first 1,000 of which were used for parameter optimization. We modeled bug reports with $K = 100$ topics for desktop, 60 for Android and 30 for iOS; we choose K based on the number of distinct words for each platform; Section 6 discusses caveats on choosing K. Finally, we labeled topics according to the most representative words and confirmed topic choices by sampling bug reports for each topic to ensure the topic's label and representative words set were appropriate.

4.2 Bug Nature and Evolution

How Bug Nature Differs Across Platforms. Table 3 shows the topics extracted from the sampled data set. We found that for desktop, application crash is the most common bug type, and application logic bugs (failure to meet requirements) are the second most popular. For Android, bugs associated with the user interface (GUI) are the most prevalent. For iOS, application logic bugs are the most prevalent.

How Bug Nature Evolves. To study macro-trends in how the nature of bugs changes over time, we analyzed topic evolution in each platform. For desktop, application logic and crashes are a perennial presence, which is unsurprising. However, while in the early years (before 2005), compilation

sets for brevity.

Table 4: Top words associated with major topics.

Label	Most representative words
Desktop	
crash	crash fail call check log process item size expect event state titl menu point block
application logic	messag updat configur link control task access thread directori cach method displai correct command modul
Android	
UI	android screen applic messag menu button text select option error fail wrong mode crash icon
thread handler	android app thread log type init intern phone zygot event handler window displai looper invok
phone call	call phone send account press devic server servic network mobil stop receiv wait confirm lock
iOS	
general	phone file call updat crash touch applic support point type menu post delet upgrad network
screen display	screen button displai view click error scroll bar game imag left load tap keyboard landscap
compilation	user run page receiv attach fail error compil mode revision map enabl crash devic handl

bugs were a popular topic, after 2005 new kinds of bugs, e.g., concurrency (topic "thread") and multimedia (topics "audio", "video") take center stage.

For Android, it is evident that in the beginning, developers were still learning how to use the platform correctly: intents are a fundamental structure for intra- and inter-app communication, and "intent" is a predominant topic in 2007 and 2008. The GUI ("UI"), concurrency ("thread handler"), and telephony ("phone call") are perennial issues.

For iOS, the GUI ("UI") and display ("screen display") are perennial issues, but in contrast to Android, concurrency and the platform do not appear to pose as much difficulty, and in later years application bugs take over. However, compilation issues seem to be a perennial problem as well, whereas for Android they are not. Table 4 shows the major topics and the top keywords within each topic.

4.3 Topics of High Severity Bugs

Our topic analysis so far has extracted topics from bug reports *at all severity levels*, from feature requests to critical bugs, which is useful for understanding the whole spectrum of maintenance: adaptive, perfective, corrective, preventive.

Table 5: Top words and topic weight for high-severity bugs.

Label	Top keywords	Weight
Desktop		
validation	build tinderbox thread widget config shell crash compon testcas loader script modul plugin nightli	31%
compilation	patch call lib make click browser access compil sourc action dialog branch trace failur displai	25%
crash	crash local remov ui control web applic button link connect request launch url render displai	22%
Android		
thread handler	android thread handler runtim zygot browser sync pointer phone menu touch tlibdvm dalvik	66%
crash	crash local remov ui control web applic button link connect request launch url render displai	23%
security	verifi warn loop execut destroi entri theme gc timer trigger similar alloc hash plugin async	5%
iOS		
crash	crash local remov ui control web applic button link connect request launch url render displai	52%
app logic	blog post phone pad publish upload save photo screen broken landscap delet refresh rotat sync	32%
make/ compile	patch call lib make click browser access compil sourc action dialog branch trace failur displai	12%

Table 6: Top-5 topics for WordPress.

Platform	# 1	# 2	# 3	# 4	# 5
Desktop	site mgmt.	style	codex	error msg.	user mgmt.
Android	null pointer	post error	upload fail	user mgmt.	codex
iOS	post error	error msg.	upload fail	landscape	codex

In addition, we are interested in finding what the show-stoppers (most pressing issues) were. Hence we extracted topics for *high-severity bugs*, i.e., bugs with severity level higher than 8 (Blocker and Critical on each platform). In Table 5 we present the major topics among critical bugs and the top keywords within each topic.

According to Table 5, in terms of the highest-weight issues, there are marked differences across platforms. For desktop, 56% of high-severity bugs are caused by validation (31%) and compilation (25%) issues. For Android, 66% of high-severity bugs are thread handler and runtime related. For iOS, application crashes account for 52% of all high severity bugs. Crashes are in fact a common high-severity bug type for the other platforms as well, 22% on desktop and 23% on Android.

4.4 Case Study: WordPress

We now focus on studying topic differences in a single app that exists on all three platforms: WordPress. We chose WordPress as our case study app for two reasons. First, it is one of the most popular blogging tools, used by 21.5% of all the websites—a content management system market share of 60.1% [31]. Second, WordPress is a cross-platform application and mature on all three platforms—desktop, Android and iOS—which reduces confounding factors (we employed the same strategy in Section 3.4).

To study differences across platforms for WordPress, we used the Section 2.4 process and set the number of topics K to 80. Table 6 shows the resulting top-5 topics for each platform. The power of topic analysis and the contrast between platforms now becomes apparent. "Post error" and "upload fail" are topics #2/#3 on Android, and #1/#3 on iOS: these are bugs due to communication errors, since spotty network connectivity is common on smartphones; "codex" (semantic error) is a hot topic across all platforms, which is unsurprising and in line with our findings from Section 4.2. For desktop, the #1 topic, "site management" is due to desktop-specific plugins and features. Since the Android version of WordPress is developed in Java, null pointer bugs stand out ("null pointer" is topic #1 in Android).

4.5 Smartphone-specific Bugs

As mentioned in Section 1, smartphone software differs substantially from desktop software in many regards: app construction, resource constraints, etc. For example, the data that smartphone software collects from rich sensors such as GPS and accelerometer raises significant privacy concerns that did not exist on the desktop platform. Furthermore, due to device portability, issues such as performance and energy bugs are significantly more important on smartphone than on fixed platforms. Understandably, significant research efforts have been dedicated to smartphone bugs such as location privacy or energy consumption. Hence we aimed to quantify the prevalence of energy, security, and performance bugs in the topic model.

We found that *energy-related* bugs (containing keywords such as "power," "battery," "energy," "drain") as a topic only ranked high (in top-5) once, in 2010 for Android—the reason was the release of Android platform version 2.2 (Froyo) in 2010, which contained a higher number of energy bugs; [7] in all other years, energy did not appear as a topic in top-20.

For *security bugs*, keywords within the topic included "security," "vulnerability," "permission," "certificate", "attack". We found that, although such bugs are marked with high severity, their representation among topics was low. We did not find them in top-5 topics; the highest was rank 7 in 2009 and rank 11 in 2010, in Android. For iOS we could not find security bugs among the top-20 topics.

For *performance bugs*, associated keywords included "performance," "slow," "latency," "lagging". We could not find performance-related topics in top-20 for Android or iOS.

5. ACTIONABLE FINDINGS

We now discuss how our findings can help point out potential improvements.

5.1 Addressing Android's Concurrency Issues

Android's GUI framework is single threaded and requires the application developer to manage threads manually, offering no thread-safety guarantees. For example, to build a responsive UI, long-running operations such as network and disk IO have to be performed in background threads and then the results posted to the UI thread's event queue; as a consequence, the happens-before relationship between GUI and other events is not enforced, leading to concurrency bugs [20]. In contrast, the iOS framework handles concurrency in a safer manner by using `GCD` (Grand Central Dispatch) to manage inter-thread communication; as a result, there are fewer concurrency bugs on iOS.

Hence there is an impetus for (1) improving the Android platform with better orchestration of concurrency, (2) improving programing practice e.g., via the use of `AsyncTask` as suggested by Lin et al. [19], and (3) constructing analyses for Android race detection [20].

5.2 Improving Android's Bug Trackers

Many of the Android projects we have examined (27 out of 38) are hosted on, and use the bug tracking facilities of, Google Code, in contrast to desktop programs, whose bugs are hosted on traditional bug trackers such as Bugzilla, JIRA and Trac. On Google Code, bug tracking is conveniently integrated with the source code repository. However, Google Code's tracker has no support for: (1) bug component—

[7] E.g., Android platform issues #8478, #9307 and #9455.

while easier for new users to file bugs as there is no need to fill in bug components, the lack of a component makes it harder for developers to locate the bug; and (2) bug resolution—they use labels instead. These aspects complicate bug management (triaging, fixing).

Hence there is an impetus for improving bug tracking in Google Code, which will in turn improve the bug fixing process for the projects it hosts.

5.3 Improving the Bug-fixing Process on All Platforms

As Figure 2 in Section 3.1 shows, the overlap between bug reporters and bug owners is higher on desktop projects. This is good for speeding up the bug-fixing process since usually bug reporters are more familiar with the bugs they report [7]. Smartphone projects' development teams should aim to increase this overlap.

According to Sections 3.2 and 3.3, Android projects have the lowest workload (highest OwnerFixed rate), and the lowest fix rate as well, which suggests a need for improving developer engagement.

The ReporterTurnover rate on Android and iOS is higher than that of desktop (Figure 3c, Section 3.2)—this indicates that there are many new users on smartphone apps, which can potentially increase product quality [6]. Hence desktop projects can improve the bug-fixing process by encouraging more users to report issues in the bug tracking system [36], e.g., via automatic bug reporting [32].

Furthermore, bug reports containing attachments, e.g., stack traces, tend to be fixed sooner [4,12]. Yet, few Android bug reports have a system trace (`logcat`) or crash report attached. Hence the Android bug-fixing process would benefit from automatically attaching the `logcat` to the bug report, which is also recommended in previous research [14].

5.4 Challenges for Projects Migrating to GitHub

For our examined period, we found that many smartphone projects have "migrated" to GitHub: 7 Android projects and 3 iOS projects have fully migrated to GitHub (source code and bug tracking), while 10 Android projects only moved the source code repositories to GitHub. [8] The rationale was developers' concern with Google Code's lack of features compared to GitHub, e.g., forks and pull requests, source code integration [2, 25]. However, the issue tracking system on GitHub lacks several critical features, e.g., severity, component (instead they only use labels); furthermore, bug reports cannot have attachments; a bug report template is missing as well. Unless GitHub adds those missing bug management features, the projects will suffer, as it is harder for developers to manage and ultimately fix the bugs.

6. THREATS TO VALIDITY

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

Selection bias. We only chose open source applications for our study, so the findings might not generalize to closed-source projects.

We studied several cross-platform projects (Chrome, Firefox, WordPress, and VLC) to control for process. However, we did not control for source code size—differences in source code size might influence features such as FixTime.

Data processing. For the topic number parameter K, finding an optimal value is an open research question. If K is too small, different topics are clustered together, if K is too large, related topics will appear as disjoint. In our case, we manually read the topics, evaluated whether the topics are distinct enough, and chose an appropriate K to yield disjoint yet self-contained topics.

Google Code does not have support for marking bugs as reopened (they show up as new bugs), whereas the other trackers do have support for it. About 5% of bugs have been reopened in desktop, and the FixTime for reopened bugs is usually high [28]. This can result in FixTime values being lower for Google Code-based projects than they would be if bug reopening tracking was supported.

IDs v. individuals. Some projects (especially large ones) have multiple individuals behind a single ID, as we showed in Section 3.2. Conversely, it is possible that a single individual operates using multiple IDs. This affects the results in cases where we assume one individual per ID.

7. RELATED WORK

Software engineering researchers have begun to explore smartphone bugs, but a study comparing bug reports/fixing processes/nature between the desktop and smartphone platforms was missing.

Smartphone bug studies. Maji et al. [17] compared defects and their effect on Android and Symbian OS. They found that development tools, web browsers and multimedia apps are the most defect-prone; and most bugs require only minor code changes. Their study was focused on the relation between source code and defects, e.g., bug density, types of code changes required for fixes. We also analyze bugs in Android Platform, but we mainly focus on bug-fixing process features. Besides Android Platform, we also consider 87 other projects on Android, desktop and iOS.

Syer et al. [29] compared 15 Android apps with 2 large desktop/server applications and 3 small Unix utilities on source code metrics and bug fixing time. We examined 38 Android projects, 34 desktop projects and 16 iOS projects. Besides fixing time, we also consider other features, e.g., severity, description length; we also analyze topics and reporter/owner trends for each platform.

Zhang et al. [35] tested three of Lehman's laws on VLC and ownCloud for desktop and Android. Their work was based on source code metrics, e.g., code churn, total commits. Our work focuses on bug reports/nature/fixing process.

Our own prior work [5] studied bug reports on Android platform and apps. The study found that for Android app bugs (especially security bugs), bug report quality is high while bug triaging is still a problem on Google Code. While there we compared bug report features across Android apps, in this work we compare bug-fixing process features across three platforms, study topics, and study feature evolution.

Topic modeling. Topic models have been used widely in software engineering research. Prior efforts have used topic model for bug localization [24], source code evolution [30], duplicate bug detection [27] and bug triaging [33].

Han et al. [11] studied how fragmentation manifests across the Android platform and found that labeled-LDA performed better than LDA for finding feature-relevant topics. Their work focused on two vendors, HTC and Motorola; we compare bug topics between desktop, Android and iOS.

Martie et al. [21] studied topic trends on Android Platform bugs. They revealed that features of Android are more problematic in a certain period. They only analyzed bug trends in the Android Platform project; in our study, we examined 87

[8] For details please visit our online supplementary material.

additional projects on Android, desktop and iOS.

8. CONCLUSIONS

We have conducted a study to understand how bugs and bug-fixing processes differ between desktop and smartphone software projects. A quantitative analysis has revealed that, at a meta level, the smartphone platforms are still maturing, though on certain bug-fixing measures they fare better than the desktop. By comparing differences due to platforms, especially within the same project, researchers and practitioners could get insights into improving products and processes. After analyzing bug nature and its evolution, it appears that build/compile processes stand to be improved on desktop/iOS, as does concurrency on Android.

9. ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation grants CNS-1064646 and CCF-1149632. This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

10. REFERENCES

[1] Investigating konqueror bugs, 2014. http://www.konqueror.org/investigatebug/.

[2] G. Art. Moving from google code to github. http://evennia.blogspot.com/2014/02/moving-from-google-code-to-github.html.

[3] A. Bachmann and A. Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *MSR'10*.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *FSE'08*, pages 308–318, 2008.

[5] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru. An empirical analysis of bug reports and bug fixing in open source android apps. In *CSMR'13*.

[6] T. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *ISSRE'13*, pages 188–197, Nov 2013.

[7] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. *CSCW'10*.

[8] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *RSSE'10*, pages 52–56, 2010.

[9] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE '13*.

[10] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *ICSE'10*, pages 495–504, 2010.

[11] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. *WCRE'12*.

[12] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE'07*, pages 34–43, 2007.

[13] C. Ikehara, J. He, and M. Crosby. Issues in implementing augmented cognition and gamification on a mobile platform. In *FAC*, volume 8027, pages 685–694. 2013.

[14] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *VLHCC'08*, pages 82–85, 2008.

[15] A. K. Karlson, S. T. Iqbal, B. Meyers, G. Ramos, K. Lee, and J. C. Tang. Mobile taskflow in context: A screenshot study of smartphone usage. In *CHI'10*.

[16] KPMG. 2013 technology industry outlook survey, 2013. http://www.kpmg.com/US/en/IssuesAndInsights/ArticlesPublications/Documents/technology-outlook-survey-2013.pdf.

[17] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *ISSRE'10*.

[18] A. Lamkanfi and S. Demeyer. Filtering bug reports for fix-time analysis. In *CSMR'12*, pages 379–384, 2012.

[19] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for android applications through refactoring. In *FSE'14*, 2014.

[20] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *PLDI'14*.

[21] L. Martie, V. Palepu, H. Sajnani, and C. Lopes. Trendy bugs: Topic trends in the android bug reports. In *MSR'12*, pages 120–123, 2012.

[22] A. K. McCallum. MALLET: A Machine Learning for Language Toolkit. http://mallet.cs.umass.edu, 2002.

[23] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *TOSEM*, 11(3), July 2002.

[24] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *ASE'11*, pages 263–272, Nov 2011.

[25] G. Rodola. Goodbye google code, i'm moving to github. http://grodola.blogspot.com/2014/05/goodbye-google-code-im-moving-to-github.html.

[26] M. Rönkkö and J. Peltonen. Software industry survey 2013, 2013. http://www.softwareindustrysurvey.org/.

[27] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE'07*.

[28] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *Emp. Soft. Eng.*, 18(5):1005–1042, 2013.

[29] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *CASCON'13*.

[30] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein. Modeling the evolution of topics in source code histories. In *MSR'11*, pages 173–182, 2011.

[31] W3Techs. Usage of content management systems for websites, 2013. http://w3techs.com/technologies/overview/content_management/all/.

[32] J. Webb. 6 third party tools for automatic bug creation and more. http://www.pivotaltracker.com/community/tracker-blog/6-third-party-tools-for-automatic-bug-creation-and-more.

[33] X. Xie, W. Zhang, Y. Yang, and Q. Wang. Dretom: Developer recommendation based on topic models for bug resolution. In *PROMISE'12*, pages 19–28, 2012.

[34] J. Xuan, H. Jiang, Z. Ren, and W. Zou. Developer prioritization in bug repositories. In *ICSE'12*.

[35] J. Zhang, S. Sagar, and E. Shihab. The evolution of mobile apps: An exploratory study. In *DeMobile'13*.

[36] M. Zhou and A. Mockus. What make long term contributors: Willingness and opportunity in oss community. In *ICSE'12*, pages 518–528, 2012.