Bug Introducing Changes: A Study with Android

Muhammad Asaduzzaman Michael C. Bullock Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{*md.asad, michael.bullock, chanchal.roy, kevin.schneider*}*@usask.ca*

***Abstract*—Changes, a rather inevitable part of software development can cause maintenance implications if they introduce bugs into the system. By isolating and characterizing these bug introducing changes it is possible to uncover potential risky source code entities or issues that produce bugs. In this paper, we mine the bug introducing changes in the Android platform by mapping bug reports to the changes that introduced the bugs. We then use the change information to look for both potential problematic parts and dynamics in development that can cause maintenance implications. We believe that the results of our study can help better manage Android software development.**

***Keywords*-Bug; fixes; bug report; change log;**

I. Introduction

Changes are unavoidable in software development and each change made to a software systems code base presents a risk that it may introduce a bug.When such a bug occurs it often goes unnoticed until someone submits a bug report describing the erroneous behaviour. Once the developers receive such a report, they evaluate it to ensure that it is a legitimate issue, and then address the issue. As part of the resolution process, developers may include an indicator about the bug they fix in the change log. Linking bug reports to the revision in which they were fixed can help us approximate which changes caused bugs. Isolating and characterizing those bug introducing changes can uncover potential problematic entities or factors that are correlated with buggy code.

In this challenge paper we mine the Android changes and bug reports [8] to uncover information about bug introducing changes. More specifically, we address the following research questions:

1) Can we identify changes that fix specific bugs?
2) Which of the system files are buggy?
3) Which days of the week are bugs introduced?
4) Does the number of files in a commit effect how likely a bug is to be introduced?

Android is of particular interest because it is an open-source system where anyone can submit changes. To make a change in Android it is a three-step process, first the change is submitted, then reviewed, and finally tested then committed if it passes. Despite this rigorous three-step process for submitting a change, Android has had over 20,000 bug reports. Our goal is to characterize problem areas for bugs in Android so those instances where making a change that is considered a high risk for containing buggy code can be completed with more caution than they might otherwise be given.

Unfortunately it is not possible to directly link a bug report to the code lines responsible for the bug as the bug report only gives a description of the problem. We use a sophisticated linking system that involves locating where the bug was likely injected into the system through analysis using the Android changes repository, the Android bug reports and the Android source trees. First, it is necessary to create a connection between the changes that potentially fix bugs to their corresponding bug reports, we call these links. To do this we look for indicators in the change logs that the change is associated to a bug report. Generally the set of links alone is not a good metric for mapping a change to a bug due to false positives. To increase the level of confidence in a link we use similarity analysis that involves matching the parts of a bug report to the parts of the linked change. Links which achieve a high level of confidence are considered to be a good link, so we call them fixes. Using the fixes it is possible to find the bug introducing code by tracking the lines involved in the fix back through the revision history. Using the GNU diff command we determine the buggy lines of code and using the Git blame command we determine when those lines were changed in the system, approximating when the buggy code was introduced. Analysis is then performed on those files for which the buggy code was introduced to characterize the injection of bugs.

The remainder of the paper is organized as follows. In section II we explain how we link a change to a bug report and confirm that it is a fix. Section III involves finding the buggy code using the fixes and approximating when those bugs were introduced. Section IV contains our findings and Section V discusses the related work. Finally, Section VI concludes this paper.

II. Mapping Changes To Fixes

In order to determine the bug introducing changes we must first determine which change(s) fixed a particular bug. We investigate both the Android bug reports and the Android change logs and with these sources it is possible to link a change which resolved a bug to its bug report by looking for common techniques developers use to report that they are addressing a bug. We then investigate those changes to confirm they are a fix.

Bug reports contain fields that allow us to filter out the reports that could not be involved in a fix prior to any analysis with the changes. For example, the bug report's type field indicates whether it is a defect or enhancement. We are interested in fixes that involve defective code so immediately we remove all bug reports that are enhancements. Also, a

bug report's status defines the current state of the bug report, some examples include new, assigned, released, spam, and duplicate. A status of released or future release indicates that the bug has been fixed and one could naively consider only those bug reports. However, while manually investigating fixes we found that some status fields are not updated to indicate that the bug has been fixed. For example, bug fixes where the status was still set as new occurred several times. Therefore, we still consider all status codes aside from those bugs that have been designated as spam or declined as it is fair to assume they will not be involved in a fix. Next, we investigate the changes to the Android platform and build the links to find fixes using the analysis techniques as described below.

A. Identifying Links

We conducted manual investigation of the Android change logs to determine in which ways developers indicate that a change involves a bug fix. We found that it is often the case that they include the URL of the bug involved, such as "Fix for http://code.google.com/p/android/issues/detail?id=15180" where 15180 would be the bug identifier (bug-id) of the bug report. It was also common practice to include the word *bug* followed by the bug-id such as "bug #: 3158459". We consider both of these cases to be strong indicators of a good match between the change and the bug report. We refer to them as a *BugMatch* link. It was also fairly common for developers to include keywords such as fixes, patch, or bug, then the bugid together on one line such as "This fixes issue 3332". We call these *KeywordNumber* matches. This type of match incorporates a higher likelihood of false positives, as there are different ways to interpret the number other than a bug-id. We filter out many of the false positives by only considering numbers after keywords that are separated by a non-alphanumeric character such as a colon or comma, still, a message line such as "Fix ULE decapsulation bug when less than 4 bytes of ULE SNDU is packed" would match, therefore we do not consider links made by *KeywordNumber* matches to be as strong as a *BugMatch*. We use the following two regular expressions to match any part of each line of a change's message:

BugMatch: $(android.*?id = |bug)\{1\}[\backslash W] * ([0 - 9]+)$

KeywordNumber:
$(fix(e[ds])?|bugs?|patch).*?\backslash W([0{-}9]+)(\backslash W|\$)+$

Each time a *KeywordNumber* finds a match we create a link between the change and the report for which the bug-id matches. However, if a *BugMatch* is found to match the change all the *KeywordNumber* links for that change are discarded as the *BugMatch* link gives us much more confidence in the link.

B. Similarity Analysis

In order to add strengths to the identified links and confirm that they are fixes we use similarity analysis on each link. Similarity analysis involves looking at the parts of a bug report and the corresponding parts of a change to find indicators that a change is a fix. Through manual inspection on randomly selected bug reports we find that high similarity rating is a very good indicator that a link is a fix. We consider the following pieces of information when completing the similarity analysis:

Bug Status. If the bug has been fixed, indicated by its status being set to either released or future release, we add 1 point to similarity rating.

People involved. There are up to four people involved in one link, an owner and a reporter in the bug report, and an author and a committer in the change. We try to match each person from the bug report to each person in the change and give +1 similarity rating for each match for a possible addition of 4 points. Bug reports only provide partial information on the people involved in the report (such as "Gir...@gmail.com"). We address this by matching a minimum the first three characters of the email address, but more if it is available.

Textual similarity. We investigate the similarity between the title of the bug report and the title of the change as well as the similarity between the description of the bug report and the message of the change using cosine similarity. For each textual similarity of 0.30 or greater we add a point to the similarity rating. We chose 0.30 by running tests on examples and found it to be a reasonable value. The bug report description can get quite long if it includes artifacts such as code or stack traces. Because of this we only consider the first 3 lines of the bug report description, as it is most likely to contain the general issue of the bug to match with the change message.

Through observation of the bug reports and changes we have determined that these criteria for similarity analysis are well suited to the Android platform. Given that there are significantly more changes than bugs it is possible we will have a good number of false positives if we simply consider that there is a link between the bug report and the change. We use the link type and similarity ratings of the link to better filter out links that are less likely to be fixes. Links that are either a *BugMatch*, or a *KeywordMatch* that has a similarity rating of at least 2 are considered as fixes.

During manual inspection a nontrivial portion of the BugMatch links appeared to be false positives, even though the BugMatch was appeared to be good. These were identified by looking at the BugMatch and doing a manual similarity analysis. One explanation for this is the possibility of multiple bug reporting tools being used that are reusing bug id numbers, another, although less likely explanation is typographical errors by those involved in making the change. On another note, the vast majority of matches with similarity analysis of 3 points or greater were positive matches. With this in hand, we suggest that similarity analysis is a crucial component of verifying links in Android.

III. Locating Bug Introducing changes

In order to detect the bug introducing changes, we first detect those revisions where bugs have been fixed. If a bug has been fixed in revision r, then we use the diff algorithm to determine those lines in revision $(r-1)$ that were either deleted or changed to fix the bug. This is accomplished by diff outputting the code chunks that are different between the two revisions in question. Clearly, because these lines fixed the bug they must have also been involved in the buggy code and as such these are the candidate lines that contain the bug. We use Git blame command (with option -C -M) to determine the date and time of the line creation. While -C finds copies, -M finds code movements [1]. We then filter out those lines that were created after the user reported the bug, since the unintended behaviour observed in the bug report can only be attributed to those lines that were created before the bug was reported. The remaining lines represent the bug introducing changes. We did not trace back through the individual revisions to look at the origin of the lines because Git tracks the history of lines even when they are moved or copied. Thus, the time stamp associated with the remaining lines indicates the date and time of their creation. We refers these lines as the buggy code.

After the buggy code lines have been identified we analyze them to attempt to identify issues in the development process in the Android platform. We first rank all the files by the number of bugs found in that file to determine which parts of Android are at a high risk of containing a bug. Next, using the blame command we identify which day the buggy code was authored to determine which days of the week present the highest risk for injecting buggy code.

IV. Results

We consider both Android bug reports and Android changes in our experiment [8]. There were 20,169 bug reports, 16,118 of which were defects, rejecting those with a status of *Spam* or *Declined* leaves 13,626 bug reports. Next, we analyze the 1,171,660 changes in the Android system. After analysis we find 48,480 links but after filtering the weaker links we find 854 fixes, 112 of which have had a similarity value of 2 or higher. We manually investigated 64 of the 112 with high similarity values and confirmed they were all bug fixes giving us confidence that a high similarity value is an indicative of a good match. We also investigated many of the *BugMatch* links and confirmed that they were also a good indicator of a bug fix giving us confidence that our filtering and matching techniques provide a high precision. By capturing only 854 fixes it may seem as if recall is very low considering there are 13,626 bug reports but we note that only 1,825 of the bug reports have a status of released or future release which indicates that the majority of bug reports have either yet to be resolved or were not considered an issue.

Table I
Top five buggy files

Rank	File Name
1	sound/pci/hda/patch_realtek.c
2	drivers/net/wireless/iwlwifi/iwl3945-base.c
3	sound/pci/hda/patch_sigmatel.c
4	drivers/ata/libata-core.c
5	drivers/md/raid5.c

Table II
Distribution of bug introducing changes by day of the week

Category	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Fixes(%)	7.89	21.28	15.18	13.99	14.13	15.03	12.50
Non-Fixes(%)	11.40	17.58	17.94	17.25	15.42	12.85	7.56
BIC(%)	7.32	18.95	18.37	10.05	8.54	34.63	2.12

A. Which files are buggy?

We believe it is important to recognize when a particular portion of the software is consistently causing problems to the execution of the system. By identifying the files which contain the most buggy code developers can examine those files more closely to better reduce the likelihood of it containing further bugs. Our intuition tells us that the files with the most bugs will be those files that are used frequently by other parts of the system. Table I indicates the top 5 buggy files as ranked by our analysis.

B. Do Android developers need to be careful on Fridays?

Śliwerski et al. [2] found that the likelihood of creating a bug introducing change is highest on Friday. However, their results were based on the data available from the MOZILLA and ECLIPSE projects. Android differs from those systems in several ways. For example, it uses a distributed source code management system and projects are associated with different groups of people. We grouped the changes by the day of the week that they were authored and found that largest number of buggy lines are introduced on SaturdayThus, the developers need to be careful about the changes that are made on that day of the week more than any other. See Table II for the breakdown of the percentage of commits over days of the week. When considering the number of fixes we find a relatively even distribution across the week with slightly fewer fixes occurring on Monday and slightly more on Tuesday. Understanding why certain days of the week cause more buggy changes than others remains as a future work.

C. Is there a correlation between large commits and buggy code?

More specifically, we wanted to know whether there is a correlation between the bug introducing changes and the number of files changed in a commit. Changing a large number of files in a single commit potentially indicates that the developer is working with different concepts and features. Comprehending and tracking large number of changes poses significant threat and could point to a haphazard development process which may result in more bug introducing changes.

Table III
NUMBER OF FILES PER COMMIT

Category	Mean	Max	Std Dev.
Fixes	1.927	33	2.46
Non-Fixes	3.123	154916	138.133
Bug Introducing	10.948	7698	121.147

Table III shows the number of files committed per change. We observe that changes that introduce bugs into the system are generally larger than the average change. This fits ones intuition regarding how larger changes impose a greater threat of introducing a bug. Also, fixes are usually very precise changes. We cannot go without mention of the very large maximum value for non-fixes, this value is the result of an initial Git repository build.

V. RELATED WORK

The term fix-inducing changes, changes that lead to problems as indicated by the fixes, was first introduced by Śliwerski et al. [2]. They used a combination of techniques to determine changes that are later altered to fix bugs. Their technique is an improvement to previous approaches [5], [6], [7]. They collected bug reports from the BUGZILLA bug tracking system and determined the changes that were made to fix those bugs by analyzing the log messages of CVS archives using a combination of syntactic and semantic analysis that we expand on. These changes, also called fixes were then traced back to determine previous changes that were made prior reporting the bugs. These are the changes that caused the later fixes. They extracted the fix-inducing changes for open-source software systems (Eclipse and MOZILLA) and reported that changes of this category have distinct characteristics. First, they found that fix-inducing transactions are large. Second, they categorized the changes based on the day of the week and found that changes made on Friday highly correlated with later fixes.

Although unique, the previous approach suffers from several limitations as pointed out by Kim et. al. [3] Not all changes are fix-inducing (such as blank lines). Moreover, the annotation information provided by the SCM system is insufficient to capture the origin of fix inducing changes, since it does not tell us which lines in one revision come from which lines in the previous revision. To resolve these limitations, they used annotation graph together with ignoring changes that does not change the behaviour of the program (such as blank lines, comments and changes results from formatting the code). They validate the effectiveness of their algorithm through a case study and pointed several applications of the algorithm. Effort has been also made to determine factors that correlate with bug introducing changes [4]. To predict those changes at the file-level, machine learning techniques have been used. The study result reveals that by predicting bug introducing changes, we can separate clean changes from the buggy lines.

Our work differs from the above approaches in that, we focus on the application of the algorithm. The Android change data set and the bug reports make an opportunity to apply the algorithm to determine bug introducing changes and to uncover issues from the changes that can guide Android developers to better manage their systems.

VI. CONCLUSION

We were able to successfully create a mapping between the Android bug reports and the Android changes to locate the changes which were bug fixes. From these fixes we identified the buggy code lines from the previous revision and isolate the date those lines were added to determine the bug introducing change.

We then mined the bug introducing changes in an attempt to understand under which circumstances buggy code is injected into the Android software system. Some of our findings conflict with those who have done research on other software systems. We found that Android bugs were most often committed on Saturday. However, our findings also fit ones intuition, we investigated the number of files that were committed in a buggy commit and attempted to understand whether the number of changes in a commit have an effect on whether or not that commit injected a bug and we found that bug introducing changes involve more files on average than fix commits. Bug introducing changes also involve more files than the non-fixes but both are associated with a high standard deviation. Finally, we looked at the files that contained the most buggy changes to identify which files authors should be most cautious about changing.

REFERENCES

[1] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git", in *MSR*, pp. 1-10, 2009.

[2] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?", in *MSR*, pp. 1-5, 2005.

[3] S. Kim, T. Zimmermann, K. Pa, and E. J. Whitehead, "Automatic Identification of Bug-Introducing Changes", in *ASE*, pp. 81-90, 2006.

[4] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?", IEEE Transactions on Software Engineering, vol. 34, no. 2, pp. 181-196, 2008.

[5] D. Cubranić and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts", in *ICSE*, pp. 408-418, 2003.

[6] M. Fisher, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for future tracking", in *WCRE*, pp. 90-99, 2003.

[7] M. Fisher, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems", in *ICSM*, pp. 23-32, 2003.

[8] E. Shihab, Y. Kamei, and P. Bhattacharya, "Mining Challenge 2012: The Android Platform", in *MSR*, 4 pp. (to appear), 2012.