An Empirical Study of Dormant Bugs

Tse-Hsun Chen[1], Meiyappan Nagappan[1], Emad Shihab[2], Ahmed E. Hassan[1]
Queen's University, Canada[1], Rochester Institute of Technology, USA[2]
{tsehsun, mei, ahmed}@cs.queensu.ca[1], emad.shihab@rit.edu[2]

ABSTRACT

Over the past decade, several research efforts have studied the quality of software systems by looking at post-release bugs. However, these studies do not account for bugs that remain dormant (i.e., introduced in a version of the software system, but are not found until much later) for years and across many versions. Such dormant bugs skew our understanding of the software quality.

In this paper we study dormant bugs against non-dormant bugs using data from 20 different open-source Apache foundation software systems. We find that 33% of the bugs introduced in a version are not reported till much later (i.e., they are reported in future versions as dormant bugs). Moreover, we find that 18.9% of the reported bugs in a version are not even introduced in that version (i.e., they are dormant bugs from prior versions). In short, the use of reported bugs to judge the quality of a specific version might be misleading. Exploring the fix process for dormant bugs, we find that they are fixed faster (median fix time of 5 days) than non-dormant bugs (median fix time of 8 days), and are fixed by more experienced developers (median commit counts of developers who fix dormant bug is 169% higher). Our results highlight that dormant bugs are different from non-dormant bugs in many perspectives and that future research in software quality should carefully study and consider dormant bugs.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software Quality Assurance (SQA)*

General Terms

Software Quality

Keywords

Software Bugs, Software Quality, Empirical Study

1. INTRODUCTION

A plethora of research efforts focus on helping practitioners discover software bugs by predicting [1, 2, 3, 4, 5, 6, 7, 8, 9], detecting [10, 11], and understanding software bugs [7, 12]. The majority of such efforts focus on studying post-release bugs [2, 6, 7, 8, 9, 5], which are bugs reported within a fixed period of time after the software is released. However, some bugs may remain dormant in a system for years, and are hence not accounted for in such studies (i.e, they will show up in a future version as dormant bugs). Moreover, some of the reported bugs might not have been introduced by the most recent version (i.e., bugs dormant from prior versions).

Despite the importance of dormant bugs, little work has been done to understand their characteristics. Therefore, in this paper we conduct an empirical study using 20 Apache software systems. The study explores the characteristics and importance of dormant bugs and compare them to that of non-dormant bugs. We define dormant bugs as *a bug that was introduced in one version (e.g., Version 1.1) of a system, yet it is NOT reported until AFTER the next immediate version (e.g., is reported against Version 1.2 or later).*

We find that 33% of the bugs introduced in a version are not reported till much later (i.e., they are reported in future versions as dormant bugs). Moreover, we find that 18.9% of the reported bugs in a version are not even introduced in that version (i.e., they are dormant bugs from prior versions). In short, the use of reported bugs to judge the quality of a specific version might be misleading. We then proceed to investigate the characteristics of such bugs along the following research questions:

RQ1: How quickly are dormant bugs fixed?
We find that dormant bugs are fixed faster than non-dormant bugs: dormant bugs have a median fix time of 5 days and non-dormant bugs have a median fix time of 8 days. We also find that dormant bugs have a statistically significant higher reopen rate than that of non-dormant bugs, even though both types of bugs are rarely reopened (90% are never reopened). This indicates the importance of dormant bugs and shows that once they are found, they are fixed quickly.

RQ2: What is the size of a dormant bug fix?
It is not clear whether dormant bugs are fixed faster because the fix is simpler or small. Therefore, we study the code changes to understand the complexity of dormant bug fixes. Surprisingly, we find that dormant bug fixes involve modifying more lines of code than non-dormant bug fixes. Dormant bug fixes have a median fix size of 19 lines of code,

whereas non-dormant bug fixes have a median fix size of 10 lines of code (almost double).

RQ3: Who fixes dormant bugs?
We find that more experienced developers are assigned to fix dormant bugs. Developers who fix non-dormant bugs have contributed a median of 278 commits, whereas developers who fix dormant bug have contributed a median of 748 commits (169% more). Once again, the fact that more senior personnel are assigned to these dormant bugs may indicate their importance.

RQ4: What are the root causes of dormant bug fixes?
Understanding the root causes of dormant bug fixes could shed light as to why experienced developers are assigned to dormant bugs. By manually studying 714 bug reports (half dormant and half non-dormant), we find that dormant bugs are mostly caused by corner cases, wrong control flows (whereas non-dormant bugs are mostly caused by incorrect function implementation), which make dormant bugs more difficult to debug, and hence may require the knowledge of experienced developers.

In summary our study highlights the unique nature and importance of dormant bugs. Future bug studies should carefully consider dormant bugs.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 describes the studied systems, our data collection approach, and defines dormant bugs. Section 4 discusses our motivation for studying dormant bugs. Section 5 presents the results of our research questions. Section 6 highlights the potential threats to validity of our study. Finally, Section 7 concludes the paper.

2. RELATED WORK

In this section, we summarize prior work on bug analysis and prediction along two dimensions: 1) discovery time, and 2) impact on software system.

Studies on Bugs that are Discovered at Different Times

Prior studies have looked at pre- and post-release bugs.

Pre-release Bugs. Pre-release bugs are bugs found before a system is released. Nagappan and Ball [13] predicted the pre-release bug density of the Windows Server 2003 using a static analysis tool. Zimmermann *et al.* [14] predicted pre-release bugs using a variety of software metrics.

Post-release Bugs. Nagappan *et al.* [4] predicted post-release bugs (which are bugs found in a fixed period of time after the system is released or in the next immediate version) using source code metrics. They used 5 different Microsoft systems to perform their case study and found that it is possible to build prediction models for an individual project, but no single model can perform well on all projects. Khomh *et al.* [15] studied how does the release cycle of Firefox affect users' perception of post-release bugs. Bird *et al.* [12] studied the relationship between code ownership and pre- and post-release bugs, and discovered that the total number of minor contributors has a negative correlation with code quality. While prior studies looked at when the bug is reported, we focus on when a bug has been introduced. In particular, dormant bugs may be reported many versions after they were initially introduced. Hence pre-release or post-release bugs may contain dormant and non-dormant bugs in them.

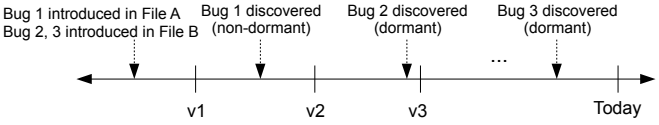


Figure 1: Definition of dormant bugs.

Studies on Bugs that have Different Impact

Bugs can also be classified by their impact on the system (e.g., high impact, or security, or performance). There are several past studies that have focused on understanding characteristics of such bugs. For example, Shihab *et al.* used different software metrics to predict high-impact bugs, and showed that these bugs have high severity [5]. Zimmermann *et al.* predicted security-related bugs in Windows Vista using classical software metrics such as churn and complexity [16]. Li *et al.* empirically studied software bugs in open source systems, and found that semantic errors are the dominant root causes of software bugs and the number of security bugs are increasing [17].

Jin *et al.* manually studied 109 performance bugs, and proposed a performance bug detection technique [10]. The authors also found that it takes a longer period of time to discover performance bugs compared to that of functional bugs. Nistor *et al.* studied the characteristics of performance bug fixes, and how performance bugs are reported [18]. All of the above studies examine bugs of a specific type based on the impact that such bugs can have on the software system. Similar to these studies, we study a specific type of bug (i.e., dormant bugs) in this paper.

Zaman *et al.* [19] compared various characteristics of performance and security related bugs. They compared the resolution time, the number of fixers and the complexity of those fixes for both, performance and security bugs. They found that the fixers of performance bugs have more experience and that the fixes involve more changes. Our study is different since we compare dormant and non-dormant bugs. We use a similar but more comprehensive set of metrics when comparing dormant and non-dormant bugs, and we conduct a manual study on their root causes.

3. DORMANT BUGS

In this section, we define dormant bugs and describe our approach to identify such bugs in our case study systems.

Definition of Dormant Bugs

We define a dormant bug as ***a bug that was introduced in one version (e.g., Version 1.1) of a system, yet it is NOT reported until AFTER the next immediate version (e.g., is reported against Version 1.2 or later)***.

Figure 1 shows an example to illustrate the definition of dormant bugs. Assuming there are three bugs being introduced in version 1 (*v1*), and Bug 1 is in File A, and Bug 2 and 3 are in File B. Bug 1 is considered a non-dormant bug, because it is discovered after *v1* but before *v2* is released. Bug 2 and 3 are both considered as dormant bugs, because they is discovered after *v2* (next immediate version of v1).

The perceived software quality may be misjudged if we only look at non-dormant bugs. For example, File A is found to be buggy in *v1*. However, File B actually contains two

Table 1: Summary of the collected issues. Issues without *affected version* are excluded in the fifth and sixth columns.

System	Language	# Bug Issues Collected	# Fixed & Resolved Bug Issues	# Dormant Bugs	# Non-Dormant Bugs	Lines of Code (K)
Abdera	Java	197	139	12	76	74
Accumulo	Java	619	471	73	129	180
Axis	C/C++	2,636	1,228	194	565	388
Bookkeeper	Java	331	237	15	83	148
Camel	Java	2,031	1,710	562	919	912
Cassandra	Java	2,984	2,055	622	888	213
Cocoon	Java	1,685	1,109	106	762	1,451
CouchDB	Erlang	989	531	184	282	176
CXF	Java	3,030	2,440	438	1,578	789
Derby	Java	3,596	2,326	450	1,618	733
Felix	Java	2,091	1,604	822	274	651
Hive	Java	2,424	1,353	151	336	785
OpenEJB	Java	552	432	103	199	520
OpenJPA	Java	1,481	932	249	502	502
Pig	Java	2,004	1,403	98	887	383
Qpid	Java	2,873	2,195	245	1,381	623
Shiro	Java	150	119	34	62	61
Thrift	Multiple	1,034	719	87	303	163
Wicket	Java	3,163	2,192	772	1,195	280
Wink	Java	192	144	4	35	137
Total	—	34,062	23,339	5,113	12,074	9,169

bugs, but they are just not yet being discovered. If Bug 2 is discovered, then *v2* may be considered to have one bug, although Bug 2 was introduced in *v1*. As a result, dormant bugs may affect how researchers and developers quantify the quality of a version. We have a detailed discussion about the importance of dormant bugs in Section 4.

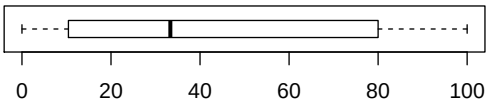
Approach to Determine Dormant Bugs

We study dormant bugs from 20 different Apache open-source systems. These 20 systems vary in size and languages (shown in Table 1), and many of them are widely used in commercial settings. Table 1 shows a summary of the JIRA issues that we crawled from the JIRA repository [20] (a bug tracking and project management system) for each of these 20 systems. In total, we crawled 34,062 JIRA issues of the type Bug, which we call bug issues (third column in Table 1). Since bug issues may contain a large amount of noise (e.g., bug issue's resolution is *Not a Problem* or *Invalid* in JIRA) [21], we only study bug issues that have a resolution of Fixed (fourth column in Table 1). Note that we use all versions of the studied systems when calculating the number of dormant bugs. The total number of dormant and non-dormant bugs that we studied are shown in the fifth and sixth column of Table 1, respectively.

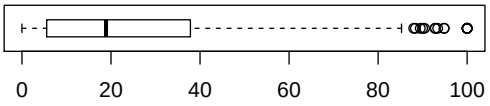
Each JIRA issue has several fields that contain important information about the issue. We use created date, version release date, and affected version in JIRA issues to determine the bug-introducing time and dormant time.

Bug-introducing Time. We use the version time of the affected version as the bug-introducing version. If a bug affects multiple versions, we use the earliest affected version among all of the affected versions to define when the bug was introduced. Since lower version numbers may not always indicate earlier versions, we use the actual version release date of the affected version to determine the release order of the versions. If a JIRA issue does not specify an affected version, we exclude it in our analysis, regardless of whether it is a dormant or non-dormant bug (about 26% of the bug issues were excluded, as shown in Table 1).

Dormant Time. When a bug issue is created in JIRA, a created date field is automatically generated. To compute the dormant time of a bug (i.e., how long has the bug been undiscovered), we use the time difference between created date and the release date of the earliest affected version.



(a) % of bugs introduced in a version but reported later.



(b) % of bugs reported in a version but introduced earlier.

Figure 2: Version-level analysis of dormant bugs.

4. MOTIVATION: WHY SHOULD WE STUDY DORMANT BUGS?

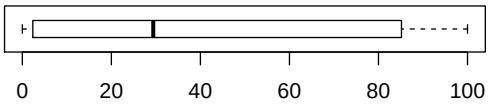
In this section, we present an initial analysis to show why dormant bugs are important. We examine the introducing and discovery time of dormant and non-dormant bugs in each version of the studied systems.

Since some bugs may not be reported till much later, using only non-dormant bugs to evaluate the overall quality of a version of a system may be inaccurate. If many bugs are reported in a version, but these bugs were introduced in previous version, then the quality of that version may be misjudged. Thus, we are interested in knowing the number of bugs that are reported in a version and which were introduced much earlier; and how many bugs that are introduced in a version and which may not be reported till much later.

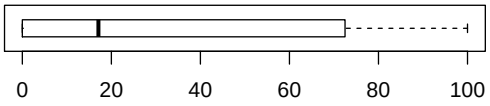
We first perform our analysis at the version level. We examine the reported and introduced bug issues in each version. We find that a median of 33% of the bugs introduced in a version are not reported till much later, and that a median of 18.9% of the bugs reported in a version are not even introduced in that version. Figure 2a is a box plot of the percentage of bugs that are introduced in one version but are not reported till later versions (i.e., bugs that appear as dormant bugs in future versions). Figure 2b shows the percentage of bugs that are reported in a version but which have been introduced in earlier version (i.e., dormant bugs). In short, the use of reported bugs (without considering dormant bugs) to judge the quality of a specific version might be misleading.

We further study the impact of dormant bugs on the perceived software quality at the file-level. In order to do this, we map the bug issues to the associated files by using the commit messages. Previous studies [22, 23, 24] have shown that the commit messages in Apache systems are usually complete and have a high quality. If a commit message contains a bug issue key, the files that are associated with the commit are related to the bug [22]. In total, we obtain 84,260 files with at least one dormant or non-dormant bug in our studied systems.

We look at files that have only dormant bugs, which implies that these files were initially "bug-free" after a version is released, but they actually contain bugs that are just not yet discovered. Figure 3a shows the distribution of the percentage of such files across the studied versions. A median of 29.4% of the files that are not considered buggy in a ver-



(a) % of files reported buggy even though bugs were introduced in earlier versions.



(b) % of files reported non-buggy (clean) even though they have a bug that will be reported in future versions.

Figure 3: File-level analysis of dormant bugs.

sion, actually contain bugs that will be reported later (i.e., are buggy files). Furthermore, we find that given a particular version, 17% of the files that are reported buggy in that version are actually bug-free files for that version (instead the bugs that they contain are dormant bugs introduced in prior versions) (Figure 3b). Our file-level analysis shows that current quality models (e.g., logistic and classification models) are most likely missing to capture the true rationale for the introduction of bugs since many of the files that are marked as buggy in a version are due to activities that occurred in much earlier versions, and files that are marked as bug-free do have bugs in them. We believe that the median between Figure 2a and Figure 2b, and the median between Figure 3a and Figure 3b are very different because a file may have cumulated dormant bugs from many previous versions.

We also find that files with only dormant bugs are found in more directories (13,524 files found in 2,990 directories, on average 4.5 files per directory) than the files with only non-dormant bugs (67,114 files found in 7,148 directories, on average 9.39 files per directory), which implies that dormant bugs are scattered across more directories and may require more testing effort across multiple components.

5. CASE STUDY RESULTS

In this section, we present each research question along three parts: the motivation of the research question, the approach that we use to address the research question, and our experimental results.

RQ1: How Quickly are Dormant Bugs Fixed?

Motivation. Bugs may exist in software systems and remain undiscovered for a long time. Such bugs often catch developers off guard. We conjuncture that dormant bugs are assigned and fixed faster than other types of bugs, because they have a higher impact on the software quality of the system [25]. Once they are discovered, such important bugs need to be addressed as early as possible (i.e., fixed faster) to reduce their negative impact. Also, it is important that the right developer is assigned to these important bugs so these bugs can be fixed quickly, without having to be tossed several times (i.e., passed around from one developer to another), which largely increases the time to fix the bug [26]. Finally, a bug needs to be fixed completely (with no re-opening) to reduce its impact on software quality.

However, if a bug has been dormant for years, it is not clear if such bugs are still considered to be important to developers. A long dormant period may simply be due to the fact that a piece of code is rarely executed, or that the number of users of that part of the system is small. Therefore, in this research question, we study the bug fix characteristics that impact the time to fix dorman bugs, and compare them to non-dormant bugs.

Approach. To answer this research question, we compute the following four software metrics. Each of the four metrics aims to represent a specific dimension of the time to fix a bug, in order to shed light on the key differences between dormant bugs and non-dormant bugs.

Fix time. Bug fix time is the time taken for a bug to be fixed once it is discovered. The bug fix time is calculated as the time period between the date when a bug issue is created till its resolution date (resolution = fixed in JIRA). For example, if a bug was reported in 2012/6/14 and was resolved on 2012/6/17, then the fix time for this bug is 3 days. We calculate the fix time for all dormant and non-dormant bugs that have at least one affected version listed in the bug issue. For the bugs that are re-opened, we calculate the entire time period until the bug is fully fixed (i.e., we aggregate the time for every fix attempt).

Number of times a bug issue is reopened. A bug issue may be reopened due to many reasons (e.g., the bug cannot be reproduced due to lack of information, the bug cannot be fixed entirely, or the fix causes other bugs [27]). Reopened bugs take longer to fix, increase maintenance costs, and can possibly affect users' perception about software quality [28]. If dormant bugs are reopened more often but are fixed faster than non-dormant bugs, then it may indicate that developers are really concerned about dormant bugs. We measure the number of times a bug is reopened by crawling the JIRA repository of all the studied systems. The life cycle of a bug issue in JIRA can change from created to closed, and may change to reopened if necessary. This metric counts the number of times a bug issue is reopened.

Number of bug tosses. Bug tossing happens when a bug is reassigned to another developer. For example, tossing may happen when the developer who was initially assigned to the bug, does not have enough knowledge about the relevant components to resolve the bug. Other reasons may be that fixing the bug requires changing different components of the system, or the developer may be busy at that time. The longer a bug gets tossed, the longer it takes to fix it. We calculate the number of bug tosses by counting how many times the assigned developers have been changed.

We exclude bug issues where the information about the assigned developer is not recorded in JIRA. Out of a total of 17,187 bug issues studied (Section III), we excluded 2,692 bug issues where no developer was assigned. In total, we ended up with 14,495 bug issues that have at least one developer assigned to them. Of which, 4,548 (31%) are dormant bugs (i.e., bugs that were introduced in a prior release and reported in their corresponding release) and 9,947 (69%) are non-dormant bugs (i.e., bugs that were introduced and reported in the same release).

Bug triage time. A bug issue needs to be reviewed and understood in order to be assigned to the right developer. The bug triage time is the review time required to change the status of a bug issue from unassigned to assigned. Bug triage time is different from bug-fixing time, because bug-fixing time includes the bug triage time, possible bug tossing time, and the time to reproduce, debug, and fix the bug.

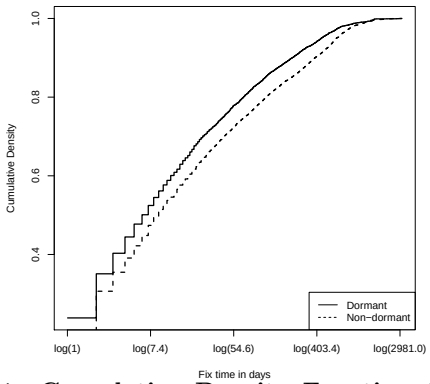


Figure 4: Cumulative Density Function (CDF) of the fix time for dormant and non-dormant bugs. The x-axis shows the number of days for a bug to be fixed in log-scale, and y-axis shows the cumulative density.

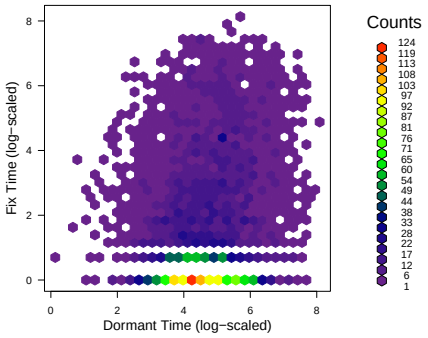


Figure 5: Hexbin plot of dormant time and fix time (in log-scale). Count represents the frequency of the data points (i.e., number of dormant bugs) that are within the range bounded by each hexagon.

Results.

Dormant bugs are fixed faster than non-dormant bugs, regardless of their dormant time. Figure 4 shows the cumulative density function (CDF) plot of the fix time for dormant and non-dormant bugs. There is about 50% chance that a dormant bug is fixed within 7.4 days, and about 50% chance that a non-dormant bug is fixed within 12 days. We log-transformed the X-axis, yet we show the unlogged values in the Figure (e.g., $log(7.4\text{days}) = 2$log-scaled days) for all CDF plots for ease of understanding. Based on Figure 4, we find that dormant bugs have a smaller fix time than non-dormant bugs (since the area under the CDF for dormant bugs is larger than the area under the CDF of non-dormant bugs). More specifically, once discovered, dormant bugs are fixed faster than non-dormant bugs.

To determine if the fix time for dormant bugs is statistically significantly smaller than that of non-dormant bugs, we use the Wilcoxon rank-sum test (also called Mann-Whitney U test) to compare the fix time of the dormant and non-dormant bugs. We choose the Wilcoxon rank test over the Student's t-test, because our dataset is highly skewed, and the Wilcoxon rank-sum is a non-parametric test which does not have any assumptions about the distribution of the sample population. A p-value ≤ 0.05 means that we can reject the null hypothesis (i.e., there is a statistically significant difference between the fix time of dormant and non-dormant bugs). By rejecting the null hypothesis, we can then accept the alternative hypothesis, which gives us statistical evidence that the values of one population are larger or smaller than the other. We set the alternative hypothesis to *greater* or *less* according to the context (e.g., we set it to *less* if we want to show that dormant bugs have a shorter fix time). We find that the p-value of the Wilcoxon rank-sum test for the fix time of dormant and non-dormant bug is $<< 0.001$. This indicates that the fix time for dormant bugs is statistically significantly smaller than that of non-dormant bugs.

Table 2 shows the mean and five-number summary of bug fix time for both dormant and non-dormant bugs. Dormant bugs have a fix time that is less than or equal to non-dormant bugs in every quantile, which further supports our finding that dormant bugs are fixed faster than non-dormant bugs.

Since dormant bugs may have a wide range of dormant times (e.g., some bugs may be dormant for years, and some bugs may dormant for months), we use a Hexbin plot to better visualize the relationship between dormant time and fix time (Figure 5). We apply a log-transformation to both dormant time and fix time to better visualize the data. In Figure 5, the data points (i.e., dormant bugs) are bounded by hexagons, and the color of the hexagon represents the frequency of the data points. A red hexagon means that there are more data points within the range, and a purple hexagon means that there are only a few data points within the range. We can see that many dormant bugs have a dormant time of around 30 to 403 days, and are fixed within a small number of days (0 to 3 days). In addition, some bugs with a very long dormant time are still being fixed within a few days. We also compute the correlation between dormant time and fix time. We find that the correlation value is very low (0.073 across all systems, and a mean correlation of -0.08 when computing the correlation for each system separately), which indicates that dormant time does not have any relationship with fix time. In other words, the length of time a bug has been dormant does not influence its fix time.

One possible reason for the fast fix time of dormant bugs is they are "surprises" to developers [5]. Thus, developers rush to fix them as soon as possible.

Dormant bugs have reopened statistically significant higher reopen counts than non-dormant bugs, although both types of bugs are rarely reopened. Figure 6 shows a CDF plot of the number of bugs that have been reopened. Both dormant and non-dormant bugs are rarely reopened, and about 90–92% of the bugs are never reopened (these bugs are all fixed and have a resolution of Fixed in JIRA). We perform a Wilcoxon rank-sum test for the number of times a bug is reopened for dormant and non-dormant bugs. The results show that dormant bugs are reopened more frequently than non-dormant bugs (p-value of 0.02).

Table 2 shows the mean and five-number summary of the reopen counts of dormant and non-dormant bugs. We see that although dormant bugs have a statistically significant larger reopen counts than non-dormant bugs, the difference is small. In fact, 99% of both dormant and non-dormant bugs were reopened *at most* once (see Figure 6).

One possible reason that dormant bugs are reopened more often is, as time passes, dormant bugs may be highly coupled with the system. Thus, fixing dormant bugs may be more complex and may cause some unexpected problems. We manually inspect 20 reopened dormant bugs to study the reasons for reopening. We found the reasons are: incomplete fixes (7), need to push the fix to other versions (5), improve

Table 2: Mean and five-number summary of metrics in RQ1 for dormant and non-dormant bugs.

	Type	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
Fix Time (days)	Dormant	79	0	1	5	42	3127
	Non-dormant	113	0	1	8	71	3081
Reopen (counts)	Dormant	0.10	0.00	0.00	0.00	0.00	4.00
	Non-dormant	0.08	0.00	0.00	0.00	0.00	3.00
Toss (counts)	Dormant	0.86	0.00	0.00	0.00	0.00	7.00
	Non-dormant	0.09	0.00	0.00	0.00	0.00	6.00
Triage (days)	Dormant	39	0	0	1	9	2923
	Non-dormant	49	0	0	1	10	3081

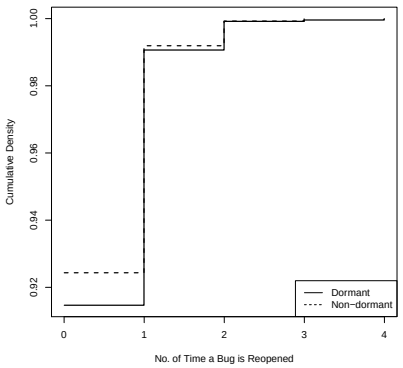


Figure 6: Cumulative Density Function (CDF) of the number of bugs reopened. The y-axis shows the cumulative density.

the fix (3), cannot reproduce the bug in the beginning (2), and the fix causes other bugs (3).

Dormant bugs do not have a statistically significant different toss rate than that of non-dormant bugs, although both types of bugs are rarely tossed. Table 2 shows that both dormant and non-dormant bugs have very a low toss rate (0 for up to the third quantile), and the distribution is similar. The result of the Wilcoxon rank-sum test shows that there is no statistically significant difference between the number of bug tosses for dormant and non-dormant bugs (p-value of 0.21).

There is no statistically significant difference between the bug triage time of dormant and non-dormant bugs. We find that both dormant and non-dormant bugs have very similar triage time distributions. In Table 2 we see that although non-dormant bugs have a slightly higher average triage time, the triage time in each quantile is very similar. The result of the Wilcoxon rank-sum test shows that there is no statistically significant difference in the triage time for dormant and non-dormant bugs (p-value of 0.80).

Dormant bugs are fixed faster than non-dormant bugs.

RQ2: What is the Size of a Dormant Bug Fix?

Motivation. In RQ1, we found that dormant bugs are fixed faster compared to non-dormant bugs. We wish to investigate whether the short fix time is due to dormant bugs having simpler (i.e., involve less files or less code) fixes.

Approach. We measure the complexity of a bug fix using the following two metrics:

- Total number of lines inserted/deleted to fix the bug,
- Total number of files modified to fix the bug,

Table 3: Mean and five-number summary of metrics in RQ2 for dormant and non-dormant bugs.

	Type	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
LOC modified	Dormant	401	0	0	19	107	955,800
	Non-dormant	572	0	0	10	100	531,800
Files modified	Dormant	7	0	1	2	5	4,032
	Non-dormant	13	0	1	2	5	5,822

The two above-mentioned metrics are used as an approximation for the complexity of bug fix in previous study [19]. To obtain the bug fix information, we crawl the Apache JIRA repository for the bug issue keys of the studied systems. We search for the issue key in the GIT commit logs, and associate the bug issue to a commit if the issue key is found in the commit log. The studied systems require developers to link GIT commits to their corresponding JIRA issues, so the links between the JIRA bug issues and their commits are fairly accurate as shown in recent work [22].

To calculate these two metrics, we count the total number of files that are modified and the total number of lines inserted and deleted for each bug fix. In the case where a bug is fixed by multiple commits, we aggregate the metric values for all the commits. For example, if one commit changes 10 files, and a second commit changes 15 files, then fixing this bug requires a modification to a total of 25 files.

Results.

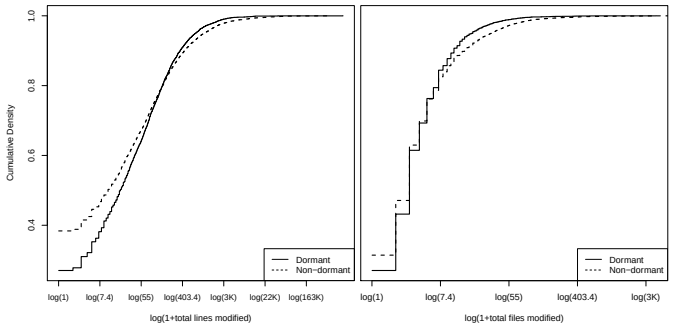
Dormant bug fixes involve modifying more lines of code and more files. Figure 7 shows the CDF of total lines of code and files modified in each bug fix, respectively, and Table 3 shows the mean and five-number summary of the metrics discussed above. We see that dormant bug fixes involve modifying a higher number of lines. However, in Table 3 we see that although dormant bug fixes modify more lines of code than non-dormant bugs, the difference is small (i.e., 9-line difference in the median) yet it is relatively large (almost double) with a median fix size of 10 vs 19 lines. We also find that both dormant and non-dormant bug fixes modify a similar number of files (the two CDF curves almost overlap in Figure 7 and the value for each quantile is the same in Table 3).

To obtain statistical evidence, we run the Wilcoxon rank-sum test, and the results show that dormant bug fixes are statistically significantly more complex (i.e., modify more lines of code and involve more files) than non-dormant bug fixes (p-value $<<$ 0.001 for total lines modified and p-value $=$ 0.001 for total files modified). Therefore, although the difference is small, dormant bug fixes are statistically significantly more complex than non-dormant bug fixes.

Dormant bug fixes are more complex in terms of lines touched and files modified, when compared to non-dormant bugs. However, the difference between number of files modified is small (median dormant bug fixes modify 9 more lines of code and same number of files than non-dormant bugs).

RQ3: Who Fixes Dormant Bugs?

Motivation. Dormant bugs are fixed faster compared to non-dormant bugs (based on RQ1), even though fixes for dormant bugs are larger (based on RQ2) . In this RQ, we investigate whether the short fix time is due to dormant bugs being assigned to more experienced developers.



(a) Total lines modified. (b) Total files modified.

Figure 7: CDF of total lines modified for fixing dormant and non-dormant bugs. X-axis shows the number of lines/files modified for fixing a bug issue in log-scale, and y-axis shows the cumulative density.

Table 4: Mean and five-number summary of experience metrics for dormant and non-dormant bugs.

	Type	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
Prior Commit	Dormant	1,676	0	56	748	2,243	12,600
	Non-dormant	973	0	0	278	1,364	12,550
Contributed JIRA issues	Dormant	468	0	9	147	486	7,076
	Non-dormant	289	0	0	62	217	6,999

Previous studies show that developers' experience is the most important factor when bugs are assigned (e.g., [29]). Since dormant bugs may have existed in the software for a long period of time, fixing such bugs may require more extensive knowledge. Experienced developers most likely posses such in-depth knowledge.

Approach. For each bug fix, we determine the developer who worked on it then look up the developer's experience at that moment in time. If multiple developers worked on a bug (e.g., due to the bug being reopened or tossed – both very rare events in our data set), we then use the average experience of all the involved developers.

Prior studies has shown that the number of commits and issues assigned to a developer can be used to derive and quantify the code ownership and experience of a developer [12, 29, 30, 31]. Hence we measure a developer's experience using the following metrics:

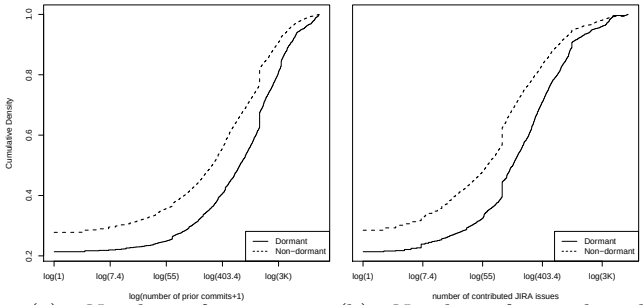
- Number of prior commits by the developer;
- Number of contributed JIRA issues by the developer.

We count all types of JIRA issues (e.g., bug, new feature, and improvement) to measure the experience of the developers, because similar to bug issues, the other types of JIRA issues can also reflect the developers' experience.

Each developer in the Apache JIRA issue repository has two different user names: a display name and a user name. The display name usually shows the actual name of the developer, and the user name usually shows the JIRA or GIT user name. However, developers may use either a display name or user name when committing changes to the GIT repository. Therefore, for each developer we obtain both his/her display name and user name from JIRA, and link the changes in GIT associated to one of the names [32].

Results.

Developers who fix dormant bugs are more experienced. Table 4 shows the mean and five-number summary



(a) Number of prior commits. (b) Number of contributed issues.

Figure 8: Cumulative Density Function (CDF) of the number of prior commits and number of JIRA issues contributed by the developers who are assigned to fix dormant and non-dormant bugs. X-axis shows the number of commits and issues resolved in log-scale, and y-axis shows the cumulative density.

of the experience metrics for dormant and non-dormant bugs. We see that in every quantile, developers who fix dormant bugs have higher experience than developers who fix non-dormant bugs. Figure 8a and 8b shows the CDF of the log-transformed total number of prior commits and contributed JIRA issues, respectively. From the CDF plots, we see that *less-experienced* developers are assigned to fix non-dormant bugs more often, while *more experienced* developers are assigned to fix dormant bugs more often.

To determine if the differences are statistically significant, we perform a Wilcoxon rank-sum test to compare the experience of these two groups of developers. For both metrics, we find that the developers who are assigned to fix dormant bugs have a statistically higher experience than those who are assigned to non-dormant bugs (both p-value $<< 0.001$).

Dormant bugs are fixed by more experienced developers.

RQ4: What are the root causes of dormant bugs?

Motivation.

Our prior RQs were primarily quantitative. They helped shed light into the dormant bug phenomena. However, we still do not have a good understanding as to why dormant bugs occur. In this RQ, we use qualitative analysis to unravel the root causes of dormant bugs and whether such causes differ from non-dormant bugs. Such in-depth understanding will assist future researchers in developing techniques to help practitioners cope and avoid dormant bugs.

Approach. We randomly sampled 357 dormant bug issues to reach a confidence level of 95% and a confidence interval of 5% [33]. We also randomly sampled the same number of non-dormant bug issues for comparison. The first author of the paper then manually examined each issue and classified them into one of the root cause categories.

We follow prior bug characterization studies [17] and classify the root cause of a bug as: *Semantic*, *Concurrency*, and *Memory* categories broadly. For the *Semantic* bugs, we further classify the bugs into one of 10 sub-categories. Table 5 shows the summary of our sub-categories. The sub-categories are borrowed from [17, 11] with modifications.

Table 6: Distribution of the manually studied dormant and non-dormant bugs among the root cause categories.

	dormant	non-dormant
concurrency	6.02	3.08
memory	1.43	1.12
semantic	92.55	95.80

For example, we include additional sub-categories that we found during our manual study and that are related to the characteristics of dormant bugs, such as *Incorrect Documentation* and *Design Issue*.

Results.

Concurrency bugs are more common in dormant bugs. Table 6 shows the distribution of the main categories of root causes for the manually studied dormant bugs and non-dormant bugs, respectively, in percentages. We find that both dormant and non-dormant bugs are mostly caused by semantic errors (92.55% and 95.8%), and that memory bugs are not very common in both types of bugs. One possible reason is that most of the studied software systems are implemented in modern programming languages (e.g., Java, and C#) that handle memory allocation issues automatically, and there are many tools for detecting memory issues (e.g., Valgrind). Concurrency bugs, on the other hand, are found in about 6% of the total studied dormant bugs and 3% of the total studied non-dormant bugs. It might be the case that such concurrency bugs tale longer to uncover since they need more varied use cases. Further in depth studies are needed to better understand this finding.

Dormant bugs are mostly caused by corner cases and wrong control flow. Since most of the studied bugs are semantic bugs, we further sub-categorize them into 10 different categories. Table 7 shows the distribution (in percentages) of the manually studied semantic dormant and non-dormant bugs in each sub-category. Semantic dormant bugs are mostly caused by *corner cases, wrong control flow, and wrong functional implementation*. We find that some *corner cases* happen only when the system is under heavy load. For example, a dormant bug in Derby was found when a user tried to create indices on a very large table (18 million rows) and with many open files [34]. Some dormant bugs are related to *design errors*. A dormant bug in Wicket was caused by an API design problem, and resulted in changes to the actual API to fix the bug [35].

Non-dormant bugs, on the other hand, are usually caused by *incorrect function implementation* (47.04%) and build problems (*Other* sub-category). For example, we find that developers in OpenJPA forgot to include a library in the system, which causes exceptions when users executing some features [36]. We also see *design errors* in non-dormant bugs, but they are less frequent than dormant bugs. In contrast to dormant bugs, *Control flow and corner case* problems are less common in non-dormant bugs.

Our manual analysis shows that dormant bugs are caused more often by control flow and corner case problems, when compared to non-dormant bugs. Since such problems are harder to debug, dormant bugs may be assigned to more experienced developers. The results from our manual study can help researchers design better dormant bug detection techniques, by focussing on specific sub-categories of root causes.

6. THREATS TO VALIDITY

We now discuss the threats to validity of our study.

Internal Validity. In this paper, we empirically study dormant bugs, and see how they differ from non-dormant bugs in terms of fix time, complexity of the fix, and developers who fix them. We do not claim any causal relationship in the findings. There may be confounding factors that influence the fix time of dormant bugs (e.g., shorter fix time may be because the dormant bug fixers are more experienced). Nevertheless, more experiments (e.g., interviews with the developers, or controlled experiment) are needed to find the actual casual relationship. In order to reduce this threat, we performed manual analysis to understand the root causes of dormant and non-dormant bugs. Moreover, some of our metrics may not always reflect the reality (e.g., larger fixes may not always be more complex).

External Validity. In this paper, we studied 20 different open-source systems. Although these open source systems have high code quality and are commonly used in commercial settings, our results may not generalize to all software systems. However, 20 case study systems are far more than typical bug prediction studies [37]. While all these systems are under the Apache Foundation, they are very varied systems (with different teams, development processes, programming languages, user base, and system type [1]).

Construct Validity.

We use information in JIRA such as affected version and created date for our metric calculations. This information is manually labeled by developers, and may contain some manual errors. However, Apache developers maintain high accuracy in the JIRA issues, and other researchers also use JIRA as a gold-set for their studies [22, 23, 24]. Additionally, if the affected version is not listed, we exclude the issue in our study (about 26% are excluded). JIRA repositories are constantly changing, and some of the issues that we studied may, for example, be reopened in the near future. However, by studying all the issues since the beginning of the development time, most of the JIRA issues that we studied are fairly stable and are not likely to change.

Since we use versions to determine whether a bug has been dormant, if the release cycles between versions are very short (e.g., a few weeks or even days), then the reported bug may be defined as dormant even though it was only introduced a few days earlier. After checking the time difference between the releases of the studied systems, we find that most systems have a release cycle of one to several months, or even a year. One exception is Wicket, in which developers may release a version within one or two weeks. These versions only fix some minor bugs introduced in earlier versions. For other systems, the time gap between versions is relatively sufficient, for users to discover and report bugs. We choose to use versions to determine dormant bugs, instead of using a fixed time period, is we want to obtain system-specific results, since the release cycle is different for each system.

Potentially we can use commands such as `git blame` to find the exact introducing time of these bugs. However, `git blame` also has it limitation (e.g., it cannot find the bug-introducing change if the change is a newly added piece of code, or design errors that cause bugs). We choose to use JIRA issues instead of using the popular heuristic-based

[1] http://www.apache.org/dev/

Table 5: Sub-categories of semantic bugs. The dimensions and categories are borrowed from [11, 17] with some modifications. Our modifications are marked using *new*.

Sub-category	Description	Abbr.
Corner Cases	Some boundary cases are considered incorrectly or ignored.	Corner
Wrong Control Flow	The control flow (sequences of function calls) is incorrectly implemented.	CtrlFlow
Design Issue (*new*)	The design of API or function is incorrect.	DesignIssue
Incorrect Documentation (*new*)	The documentation of the software is incorrect or inconsistent with the code.	IncDoc
Exception Handling	Do not have proper exception handling.	ExceptHandle
Missing Cases	A case in a functionality is not implemented.	MissC
Missing Features	A feature is supposed to be but is not implemented.	MissF
Processing	Processing such as evaluation of expressions and equations is incorrect.	Process
Typo	Typographical mistakes.	Typo
Other Wrong Functional Implementation	Any other semantic bug that does not meet the design requirement.	FuncImpl

Table 7: Distribution (in percentage) of the manually studied semantic dormant and non-dormant bugs in each sub-category.

	Corner	CtrlFlow	DesignIssue	ExceptHandle	IncorrectDoc	MissCases	MissFeatures	Processing	Typo	FuncImpl	Other
Dormant	25.08	26.01	6.50	2.79	4.02	0.31	2.48	4.64	3.72	24.15	0.31
Non-dormant	3.49	7.26	5.65	4.84	2.96	0.54	1.08	1.34	0.00	47.04	17.74

SZZ algorithm [38] because information in JIRA is manually entered by experts, which contain less noise [22]. In addition, SZZ only works when the newly added code is buggy, and not when the missing changes cause a bug. We plan to compare the dormant time calculated using our approach with the dormant time calculated using `git blame` in the future.

Our experiment results are based only on the fixed bugs, so there may be more dormant bugs in the systems. This problem exists in all studies that study bug reports. However, we are the first to highlight the importance of missing such data.

Prior studies suggested reporting *effect sizes* along with *t-tests* to quantify the difference between two populations [39]. We found that except the experience of the fixers, all other metrics have a small to trivial effect size. Nevertheless, the differences are statistically significant.

7. CONCLUSION AND FUTURE WORK

Dormant bugs are bugs that were introduced in earlier versions of a system but not found until much later. Little is known about such bugs and their impact on studies of software quality. We find that a median of 33% of the bugs introduced in a version are not reported till much later (i.e., they are reported in future versions as dormant bugs). Moreover, we find that a median of 18.9% of the reported bugs in a version are not even introduced in that version (i.e., they are dormant bugs from prior versions). In short, the use of reported bugs to judge the quality of a specific version might be misleading. Our analysis of such dormant bugs at the file-level leads to similar conclusions about the risks of not considering such dormant bugs when modeling the quality of source code files.

We empirically study the characteristics of dormant bug fixes along two different dimensions:

1. Bug fix time: we found that dormant bugs take less time to fix and are reopened more frequently compared to non-dormant bugs.
2. The personnel assigned to fix dormant bugs: we found that more experienced developers are assigned to fix dormant bugs.

On further analysis, we found that dormant bugs fixes are more complex (i.e., modify more files and more lines of code) than non-dormant bugs. Through a manual in-depth analysis of the root causes of dormant bugs, we find that they are mainly related to corner cases, wrong control flows, while non-dormant bugs are not (they are mostly related to incorrect functional implementations). In the future, we plan to study the impact of missing to consider dormant bugs when building bug prediction models. We also plan to study how we can help developers avoid dormant bugs.

8. REFERENCES

[1] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, 2005.

[2] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, 2007.

[3] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *ICSE '09*, pages 78–88, 2009.

[4] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE '06*, pages 452–461, 2006.

[5] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 300–310, 2011.

[6] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, 2010.

[7] Tse-Hsun Chen, S. W. Thomas, Meiyappan Nagappan, and A.E. Hassan. Explaining software defects using topic models. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, MSR '12, 2012.

[8] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale

experiment on data vs. domain vs. process. In *ESEC/FSE '09*, 2009.

[9] Adrian SchrŽter, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. If your bug database could talk. In *Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters*, pages 18–20, 2006.

[10] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[11] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[12] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *SIGSOFT/FSE '11*, pages 4–14, 2011.

[13] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05*, pages 580–586, 2005.

[14] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 9, 2007.

[15] F. Khomh, T. Dhaliwal, Ying Zou, and B. Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *Proceedings of the 9th International Working Conference on Mining Software Repositories*, pages 179–188, June 2012.

[16] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 421–428, 2010.

[17] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006.

[18] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *MSR '13*, May 2013.

[19] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: a case study on firefox. In *MSR '11*, pages 93–102, 2011.

[20] Apache Software Foundation. Jira. `https://issues.apache.org/jira/`, 2013.

[21] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *ICSE '11*, pages 481–490, 2011.

[22] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *SIGSOFT/FSE '11*, FSE '12, 2012.

[23] Daryl Posnett, Abram Hindle, and Prem Devanbu. Got issues? do new features and code improvements affect defects? In *WCRE '11*, pages 211–215, 2011.

[24] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. Sample size vs. bias in defect prediction. In *ESEC/FSE 2013*, pages 147–157, 2013.

[25] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[26] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *SIGSOFT/FSE '09*, ESEC/FSE '09, pages 111–120, 2009.

[27] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *ESEC/FSE '11*, pages 26–36, 2011.

[28] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 2012.

[29] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02*, pages 503–512, 2002.

[30] H. Kagdi, M. Hammad, and J.I. Maletic. Who can help me with this source code change? In *Proceedings of the 24th International Conference on Software Maintenance*, ICSM '08, pages 157–166, 2008.

[31] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *ICSE '11*, pages 491–500, 2011.

[32] Ran Tang, Ahmed E. Hassan, and Ying Zou. Techniques for identifying the country origin of mailing list participants. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 36–40, 2009.

[33] S. Boslaugh and P.A. Watters. *Statistics in a Nutshell: A Desktop Quick Reference*. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[34] Apache Derby. Derby-1661. `https://issues.apache.org/jira/browse/DERBY-1661`, 2013.

[35] Apache Wicket. Wicket-4161. 2013.

[36] Apache Felix. Felix-1460. `https://issues.apache.org/jira/browse/FELIX-1460`, 2013.

[37] Emad Shihab. *An Exploration of Challenges Limiting Pragmatic Software Defect Prediction*. PhD thesis, 2012.

[38] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05*, pages 1–5, 2005.

[39] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. Systematic review: A systematic review of effect size in software engineering experiments. *Inf. Softw. Technol.*, 49(11-12):1073–1086, November 2007.