Who introduced this bug? It may not have been caused by the previous commit!

Gema Rodríguez-Pérez
GSyC/LibreSoft
Universidad Rey Juan Carlos
Madrid, Spain
gerope@libresoft.es

Jesus M. Gonzalez-Barahona
GSyC/LibreSoft
Universidad Rey Juan Carlos
Madrid, Spain
jgb@gsyc.es

Gregorio Robles
GSyC/LibreSoft
Universidad Rey Juan Carlos
Madrid, Spain
grex@gsyc.urjc.es

ABSTRACT

The assumption that "a given bug was introduced by the lines of code that were modified to fix it" seems at first glance very reasonable. In fact, many studies on bug fixing are built upon it. However, there is little empirical evidence supporting it, and a careful examination shows other possible sources for the introduction of bugs, such as an older modification, or a change in some API in a different part of the code.

This paper presents an observational study designed to shed some more light in this area. For that, we studied the lines changed by bug fixes as a part of "the previous commit" (or commits) in the OpenStack project. Using information from the code management, issue tracking, and code review systems, we analyzed if the code introduced by previous commits was correct at the time of introduction, and was therefore the cause of the bug. Our results show that the assumption that bugs were introduced in the previous commit does not hold for a large fraction (at least 37%) of the analyzed bugs.

Keywords

Bug introduction, bug seeding, SZZ algorithm, previous commit

1. INTRODUCTION

When a failure is found in some software, developers try to fix it by locating and modifying the source code line(s) that are the cause for the wrong behavior. It may seem reasonable to assume that previous modifications of these lines are the cause of the bug. Those previous modifications are what will refer through this paper as the *previous commits*.

However, to find where and when a bug was introduced in the source code is not a trivial task, and it may be much more complex than what this assumption suggests. This fact has been largely ignored in much of the bug-fix literature, mainly because the data related to the origin of a bug is *embedded* in the evolution of the software [19], as Sinha et al. state. With this they mean that there is no easy evidence (artifact, comment or log) where developers specify what produced that error from a more historical point of view. The explanation of the cause is thus *embedded* in the project.

This is the reason why many studies in the area of software maintenance and evolution start with this implicit assumption. As an anecdotal evidence, we have found the following rationales in several research areas:

- bug seeding studies, e.g., *"This earlier change is the one that caused the later fixed"* [23] or *"The lines affected in the process of fixing a bug are the same one that originated or seeded that bug"* [12],
- bug fix patterns, e.g., *"The version before the bug fix revision is the bug version"* [17],
- defect prediction studies, e.g., *"A line that is deleted or changed by a bug-fixing change is a faulty line"* [1],
- tools that prevent future bugs, e.g., *"We assume that a change/commit is buggy if its modifications has been later altered by a bug-fix commit"* [6].

But although the assumption can be found frequently in the research literature, in our opinion there is not enough empirical evidence supporting it. That is the reason why we have conducted an observational study on bug fixing, devoting a significant effort to locate the origin of bugs in source code, and to understand the possible causes. Our main aim was to identify it the previous commit to to a bug fix was buggy "at the time of committing it". That is, if the program showed the bug at that moment because of the code introduced in that commit, or on the contrary, if the bug was caused by some earlier commit to that code, or by some other commit affecting other parts of the program.

To illustrate a case in which the assumption of the previous commit is fulfilled, Figure 1 shows an example of what we understand as a previous commit causing a bug. Let's assume that we have three different versions of the same file in the history of the control version of the project:

1. The code on the left (subfigure *(1)*) is the one after the bug was fixed.
2. The code in the middle (subfigure *(2)*) shows the previous version of the code (the previous commit), which was already causing malfunction.

ICSME '16 Raleigh, North Carolina USA

DOI: 10.475/123_4

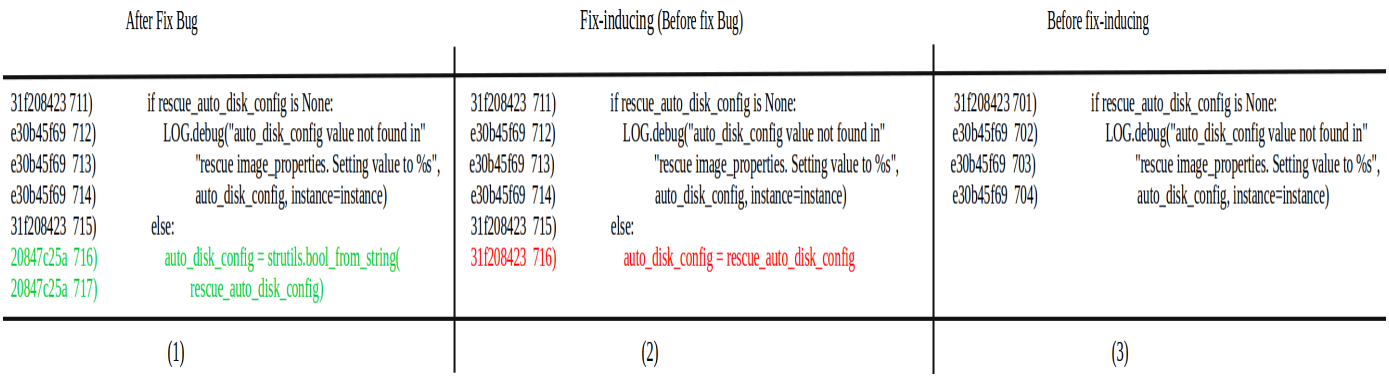

Figure 1: Example of a change in which the bug was introduced in the previous commit. More recent versions of the code are on the left.

3. The code on the right (subfigure *(3)*) shows the version of the file before the previous commit. In this case, at that point the bug did not exist.

```
Xen: convert image auto_disk_config value to bool before compare

During rescue mode the auto_disk_config value is pulled from the rescue
image if provided.  The value is a string but it was being used as a
boolean in an 'if' statement, leading it to be True when it shouldn't
be.  This converts it to a boolean value before comparison.

Change-Id: Ib7ffcab235ead0e770800d33c4c7cff131ca99f5
Closes-bug: 1481078
```

Figure 2: Description of the bug-fix commit for a case in which the previous commit caused the bug.

According to the description in the log of the commit that fixed the bug (see Figure 2), the previous commit introduced it: a string variable was produced instead of a Boolean variable, which was needed to satisfy expectations by the rest of the code.

For an example of the contrary, when the assumption is not fulfilled, consider Figure 3, which shows a case in which the code introduced by the previous commit was correct at the moment of introducing it. Now, the log of the commit fixing the bug (Figure 4) describes that the name of an argument had changed due to some other commit to other part of the software, causing the failure. In this case, the failure has nothing to do with the previous commit, but with later changes in other areas that were not properly reflected in this code. When the lines were introduced by the previous commit, they where not buggy.

This is an example of how in projects that are evolving, code that at some point was correct could become buggy later. Changes in other parts of the code may trigger bugs in places which were correct in the past. This happens often in situations of changes in called APIs: when the code was written, it was correct, although later modifications to APIs had as a side effect that the formerly correct code started to show a wrong behavior, making the software fail. In such cases, the source of the error cannot be attributed to the changes performed in the previous commit, which were correct when they were introduced, since in that moment they referred to a different API.

The goal of this paper is to find out to which extent the cause of bugs can be attributed to the previous commit. We will consider that the previous commit is the cause for the bug if that code was buggy (caused the malfunction) in the context of the code at the moment it was introduced. If the code was right at that time, but the bug is due to some other change in the chain of previous commits, or to changes to other areas of the code (such as a change in APIs), we do not consider that change to be the cause of the bug.

More formally, we address the following research questions:

- RQ1: How can we identify when changes to code do fix a bug?
- RQ2: How often is the previous commit the cause of a bug?

RQ2 is the main question that we want to answer: given bugs that have been fixed by changing some code, how many of those were introduced by the previous commit?

In practice, to answer RQ2 we first need to study the issue-tracking system and identify the subset of closed tickets that correspond to fixed bugs. In essence, RQ1 could be also stated as *Which tickets in the issue tracking system are (real) bug reports?*, assuming that we can track tickets to changes in code. This is because (real) bugs are managed in an issue-tracking system together with feature requests, optimization, test cases, etc. As we are only interested in bugs, we need to first identify those as a previous step to analyze if they have been caused by the previous commit.

While RQ1 is instrumental, with RQ2 we address a very fundamental aspect of the studies on how bugs are fixed: the underlying assumption that the commit previous to a bug fix *touched* the same lines introducing the bug. If evidence is found that in a large fraction of the cases the corresponding code was correct when it was introduced, there is no reason to blame it as the cause of the bug, even if changing it fixes the bug. Therefore, any result obtained after this assumption should be revisited with some care.

The remainder of this paper is structured as follows. Next, we present the current body of knowledge in Section 2. Section 3 describes the methodology used to identify the moment in which the bug was introduced in the source code, followed by the results obtained after applying our approach to a selection of OpenStack bug fixes in Section 5. Section 6 answers the research questions and discusses potential applications and improvements of our approach. After reporting

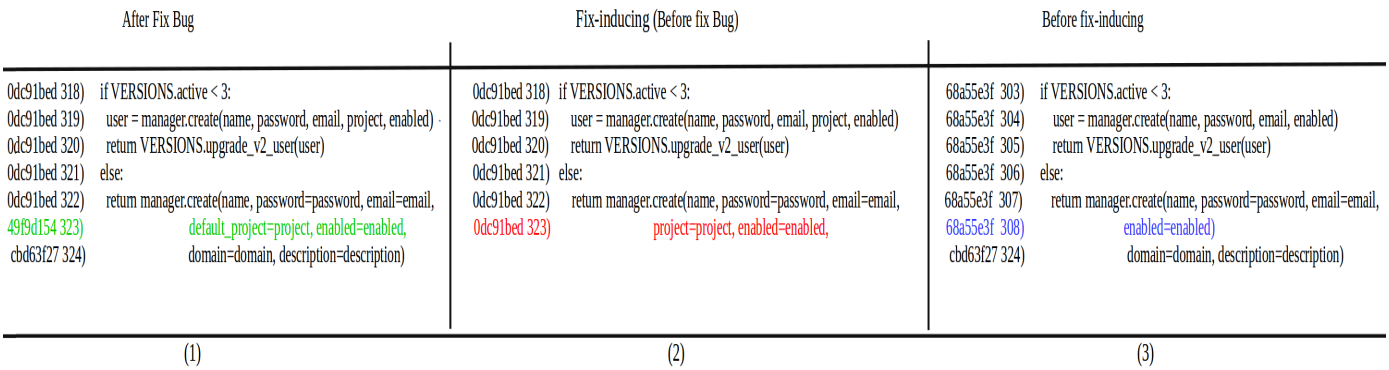

Figure 3: Example of a change where the previous commit, 0dc91bed, did not insert the bug. More recent versions of the code are on the left.

Update default_project param on create user

In keystone v3, the parameter to create user for the the default project has changed from project to default_project and is no longer honored and throws an exception. Also passing in '' rather than None causes keystone issues, so moving to None.

Closes-Bug: #1478143
Change-Id: I73423433a42bf46769065a269a3c35f27175f185

Figure 4: Description of the bug-fix commit for a case in which the previous commit did not cause the bug.

the limitations and threats to validity in Section 7, we draw some conclusions and point out some potential future work in Section 8.

2. RELATED WORK

The study of code changes introducing bugs has been a *hot* topic in the mining software repositories research community for over a decade. The most used algorithm to automatically identify them has been proposed by Sliwerski et al. [20], which is an improvement to previous approaches [4, 7, 8]. Currently, it is a well-known algorithm, called SZZ, based on text differences to discover modified, added and deleted lines between the bug-fix and its previous version. The SZZ algorithm uses the CVS `annotate` command[1] to identify the last commit that *touched* these lines.

There are several research articles that suggest improvements to the SZZ algorithm. Kim et al. [16] use annotation graphs instead of CVS annotation to locate, in the previous versions, the lines affected by modification and deletion. Also, they avoid some false positives by not considering blank spaces, changes in the format or changes in the comments. Williams et al. have revisited the SZZ algorithm to track fix-inducing changes and identify types of changes [23].

The SZZ algorithm (and its *successors*) have had a considerable impact in the research community. Noteworthy is the fact that the paper with original the SZZ algorithm [20] has been cited, according to Google Scholar, 475 times as of April 2016. The enhanced version of the SZZ algorithm [16] counts with 123 citations. Hence, there is a myriad of articles that use SZZ on an ample number of scenarios. Without trying to be exhaustive, we offer several examples. Therefore, Yang et al. apply SZZ to find what kind of bug-inducing changes are likely to become a great threat after being marked as bug-fix changes [24]. Zimmermann et al. use it for predicting bugs in large software systems [26]. Kim et al. show how to classify file changes as buggy or clean using change information features and source code terms [15]. Kamei et al. apply it to validate effort-aware bug-prediction models [13]. Eyolfson use it to study if time of the day and developer experience affect the probability of a commit to introduce a bug [5]. Izquierdo et al. use the SZZ algorithm to see if developers are fixing their own bugs [12]. Yin et al. use SZZ to find how many fixes to bugs introduce new bugs [25]. Tantithamthavorn et al. employ it to quickly identify the location of a bug [22]. Fejzer et al. use it to support code review [6]. Altman et al. use it to predict whether a change is buggy at the time of a commit, evaluating the change classification techniques to address the incorrect evaluation presented by cross-validation [1]. Asaduzzaman et al. apply the SZZ algorithm on Android to study its maintainability [3].

Prechelt and Pepper offer a good overview of the limitations of bug-introducing code changes when adopted by practitioners. They point out that one of the obstacles in the way of a reliable analysis are the "additional changes between defect insertion time and defect correction time that happen to happen at subsequently defect-corrected locations" [18]. Some methods that consider sources of information other than the previous commit have been proposed already; so, German et al. [9] point out that software is in constant change, and that changes performed may have impact across the whole system and may lead to the manifestation of bugs in unchanged parts. In this case, a bug emerges in a different location from the source of the bug, which is a change to a function somewhere else in the source code base.

Kamei et al. present a change risk model in which several characteristics of a software change are considered (lines added, developer experience...) and empirically evaluate their approach to identify in real-time those software changes that could become a defect [14]. Altman et al. use the same concept to study where authors that applied change classification on a proprietary code base, sharing the experience

[1]Other versioning systems provide similar functionality to CVS `annotate`; for instance, git offers `blame`.

and lessons learned. They concluded that interpretable prediction models are needed for software defect prediction [1].

Sinha et al. present another technique to identify the origins of a bug in [19]. Their technique is not a text-based technique like the SZZ algorithm, as the authors analyze the effects of bug-fix changes on program dependencies. So, taking into account the semantics of the source code they achieved higher accuracy in identifying the origins of a bug. The two approaches have however some methodological patterns in common:

1. They find the differences between the bug-fix version and the previous version of the file to recognize those changes done by the bug-fix commit.
2. They look back in the code revision history until they identify which version touched the lines affected in the bug-fix for the last time.

On the other hand, the question of whether a bug report is really a bug or not has been widely studied in the software maintenance and evolution research literature in the last years. In this sense, Pan et al. classify the different types of bug-fix patterns that exist most commonly [17]. Antoniol et al. have used a text-based approach to classify bug reports into corrective maintenance and other kinds of activities [2]. Herzig et al. have done a manual examination of thousands of bug reports and have found that around a third of them are misclassified, in the sense that they were not a bug report but a feature request, an internal refactoring or an update to documentation [10]. They show that almost 40% of the files marked as defective did not have a bug, and suggest that this missclassification introduces bias in bug prediction models and tools. Tantithamthavorn et al. investigated the nature of mislabeled issue reports and how they impact in the prediction models [21].

3. METHODOLOGY

In the case of OpenStack, the data needed to analyze when a bug was introduced can be obtained from the source code management, issue tracking, and code review systems, as in many other free/open source software (FOSS) projects. In our analysis, we have focused on Git[2] as source code management, Launchpad[3] as issue tracking system, and Gerrit[4] as code review supporting tool, as those are the one used by OpenStack, but our methodology should be adaptable to any other similar tools.

Launchpad works with issue reports called tickets, which describe bug reports, feature requests, maintenance tickets, and even design discussions. In our study, however, we are only interested in those tickets that have following properties:

1. They describe a bug report, and
2. They have been closed and merged in the code source to fix the described bug.

In these bug reports we can find a comment with the link to Gerrit where the bug was fixed. It is in Gerrit where we can see all the patchsets proposed and the comments done by the reviewers.

[2]https://git-scm.com/
[3]https://launchpad.net/
[4]https://www.gerritcodereview.com/

3.1 First Stage: Filtering

First, we have to identify what issues found in Launchpad are bug reports. This is not a trivial task and is labor intensive as it has to be done manually. As the process is repetitive, we developed a web-based tool[5] that helps in the classification process. This tool offers all relevant information required to decide if an issue corresponds to a bug report or not. The tool uses information extracted automatically from the project repositories, and offers a web-based interface which allows for collaboration, traceability and transparency in the identification of bug reports.

During the identification of the issues, we have to take into account the following parameters for each ticket:

- The title of the issue report
- The description of the issue report
- The description of the fix commit
- And sometimes the changes to the source code, when neither the description nor the comments by developers and reviewers in Launchpad and Gerrit clarified the underlying ticket.

A screenshot of the web interface of the tool is shown in Figure 5. The left side is used to display the information extracted from Launchpad and Gerrit; on the right researchers can write and classify the ticket into one of the three groups. Additional meta-data, such as keywords, comments and the reviewer are included in the database.

Each ticket was categorized into one of following three groups:

1. Group 1 (*Bug Report*): The ticket describes a bug report.
2. Group 2 (*Not Bug Report*): The ticket describes a feature, an optimization code, changes in test files or other situations, but not bug reports.
3. Group 3 (*Undecided*): The ticket presents a vague description and cannot be classified without doubts.

From the experience of analyzing a small number of tickets, we agreed on following four criteria:

1. Each time the title or the description of a ticket describes an unexpected behavior in the program, it was considered as a bug report.
2. If the description of the ticket presents an optimization, deletion of a dead code or the implementation of new characteristics, we agreed not to classify it as a bug report because there is no failure.
3. When the ticket described that some updates in the code were required, such as updates in the OS or in a version of a package/module which is being used in the code, the ticket is a bug report. We consider all tickets that require updating as bug reports, because updating a software hints to the software not operating as expected.

[5]bugtracking.libresoft.es

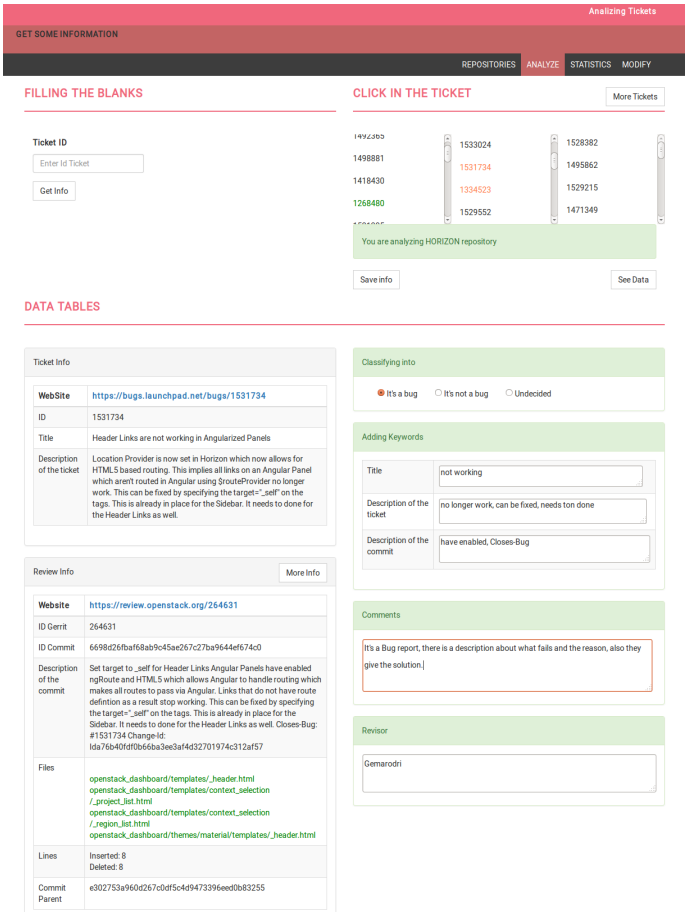

Figure 5: Screenshot of the tool used to classify the tickets.

4. When only test files are affected by a ticket, it is not a bug report. We consider bug errors in test files to be of a different type, as the software may still work as expected.

Sometimes we have been unable to classify a bug report because we did not have sufficient data or because of the complexity of the issue. In those cases, tickets have been classified into the *Undecided* group.

3.2 Second Stage: Who caused the Bug?

In this second part, our work was focused on analyzing the previous commit exclusively for those tickets classified in the *Bug Report* group. Therefore we had to locate the line that contained the bug, inquire the reason of the software failure, and sometimes gather additional information from the context of the bug and the project.

For that, we had to analyze the lines involved in the bug fix and in the *parent* commit of the bug fix commit, being sure that the lines were added, inserted or modified in the previous commit. We refer to *parent* commit as the commit that modified any line of code in the file before the fix-bug commit, in contrast to the *previous* commit where the modified lines were the same than in the fix-bug commit. It should be noted that those lines modified in the parent commit do not have to be the ones that have been modified in the bug-fix. Figure 6 contains a snapshot of the information provided by Gerrit, where the link to the parent commit(s) can be found, that corresponds to the bug-fix shown in 1. As can be seen, the previous commit (*31f08423*) in Figure 1 is different from the parent commit displayed in Figure 6 (*db7fc59ebc*).

We do this process to be sure that we are looking at the correct change, because sometimes, although the commit added many lines, the code before the commit contained some of the lines added; and such cases, we have false positives where the previous commit did not cause the bug.

Author	Andrew Laski <andrew.laski@rackspace.com>	Aug 3, 2015 10:29 PM
Committer	Andrew Laski <andrew.laski@rackspace.com>	Aug 3, 2015 10:29 PM
Commit	20847c25a8157a10b765387ff8dbda31f8f4e91a (gitweb)	
Parent(s)	db7fc595ebc86b19ead193a3571e4db2ba8de8f5 (gitweb)	
Change-Id	Ib7ffcab235ead0e770800d33c4c7cff131ca99f5	

Figure 6: Information about the bug as displayed by Gerrit.

The analysis was done manually. We used *git blame* to see the previous commit for each line of the involved file. Also, we used *diff* to see the differences between the two files, in our case as the file is going to be the same, between the file in two different moments in the control version system.

The procedure for each file involved in a bug fix is as follows:

1. git checkout *commit that fixed the bug*, git blame *file involved*. In this step we can see the lines added, modified or deleted by the commit that fixed the bug.

2. git checkout *parent of commit that fixes the bug*, git blame *file involved*. In this step we can see the previous commits for the different lines *touched* in the fixed bug.

3. git checkout *parent of previous commit*, git blame *file involved*. With this step we can ensure that the previous commit introduced these lines.

Finally we had to discard some *noise* present in our results. This happened when the changes in the previous commit could not have caused the bug. Therefore, we deleted the previous commits for which the following criteria were met:

- Exclusively blank lines
- Changes in the format
- Copied lines
- Changes in source code comments
- Updates in the version number of a file/software
- Commits that were committed after the bug report was opened in Launchpad, according to Śliwerski et al. [20].

4. EVALUATION

We have validated our methodology analyzing tickets from OpenStack. OpenStack is a cloud computing platform with a huge developing community (more than 5,000 developers) and significant industrial support from several major companies such as Red Hat, Intel, IBM, HP, etc. OpenStack was

particularly of interest because it is continuously evolving due to its very active community. Currently it has more than 233,000 commits with more than 2 million lines of code[6]. All its history is saved and available in a version control system[7], as well as its issue tracking system (Launchpad[8]) and the source code review system (Gerrit[9]).

OpenStack is composed of 9 projects, but we only focused on the main four: Nova, Cinder, Neutron and Horizon. As can be seen in Table 1, these projects have been very active during their entire history, and in the last year.

	All History	Last Year (2015)
Nova	14,558	3,283
Fuel	9,139	5,123
Neutron	8,452	3,855
Horizon	4,871	1,994
Cinder	4,556	1,832
Keystone	4,874	1,795
Heat	6,395	2,372
Glance	2,651	723
Tempest	4,141	1,312

Table 1: Commits per Project in OpenStack

For these four projects we analyzed if bug fixes where introduce in their previous commits. For the first stage, we used the tool described in 3.1. Three different researchers, were involved in this stage, resulting that each ticket was analyzed by two of the researchers independently. The second stage was done manually by the first author.

5. RESULTS

A total of 459 different tickets from the Launchpad of the four main projects in OpenStack: 125 tickets from Nova, 125 tickets from cinder, 125 tickets from Horizon and 84 tickets from Neutron.

5.1 First Stage

We classify a total of 459 tickets using the tool, resulting in 917 reviews[10]. Only those tickets classified as bug reports with a percentage of agreement of 100% by both researchers were considered in the next stage, which analyzes if the cause of the bug was introduced in their previous commits. Thus, to measure agreement among researchers in the context of a content analysis, we have used the percent agreement variable [**?**]. This process requires manual inspection by researchers. In the mean, classifying a ticket takes around 5 minutes, although the amount of time decreases with experience as could be expected. We have measured in Github the mean spends analyzing a ticket per each researcher, obtaining in the best case 3.35 minutes while 4.09 and 4.12 for the other two. Whereas analyzing who caused the bug, in mean, takes around 16 minutes per bug report.

Table 2 shows the classification percentages of the analyzed tickets for each researcher, and the number of tickets classified into the same group by two different researchers. As a result, researchers identified 292 tickets in the same group, that is, their results matched in over 70% of the cases. Of those, 209 tickets had been classified in the *Bug report* group, 74 in the *Not Bug Report* group, and 9 tickets classified in the *Undecided* group.

We also measured the concordance in the classification of each developer according to the project analyzed (see table 3). Values obtained by the three researchers are very similar, in general around a 70%. The concordance values were always above 60%.

RQ1: Using all the information available in the bug tracking system and code review systems related to a bug-fix, we have obtained that in at least 72% of the tickets analyzed the bug-fixes were real bug reports.

5.2 Second Stage

In this stage we have analyzed the 209 tickets, the possible outcome of the analysis was one of the following three options:

- Cause
- No Cause
- Undecided

This analysis takes into account that the bug could span many lines that may belong to several previous commits, but in fact, not all of them may have caused the bug. It may happen that in the previous commit lines may have been copied from further previous commits, comments may have been modified, or blank spaces/lines may have been introduced. Hence, the cause could be found in a single previous commit, in many or even in none.

Figure 7 contains a real example of a previous commit where more than one commit has been identified. In this case, we have two possible commits: *e7be0a988* and *e5296c1da*. Previous commit *e7be0a988* did not cause the bug, because the modification affects only the version number of the software. It is previous commit *e5296c1da* the one that caused the bug, because it introduced an incorrect `break` line.

We have identified a total of 462 previous commits which could be the cause of the 209 bug reports under analysis. Then, we analyzed the bug reports together with their previous commits discarding the cases where the previous commit was a false positive (*noise*) such as blank lines, changes in comments, commits added previously date of bug report or even a change in the version of the file. Therefore, in total we have analyzed 349 previous commits.

As can be seen in Table 4, from the 349 previous commits, 179 have been considered to be the cause of the bug, whereas 129 have been identified as not being the cause.

We have been unable to decide in 39 cases. Figure 8 provides an example of such a situation; the file content after the bug fix can be seen on the left in subfigure (1) and the file before the fix commit on the right in subfigure (2); as it can be observed, the only difference is lines that have been added. Figure 9 offers an example of a different situation, where the researchers have not been able to classify the bug due to not having sufficient knowledge of the change, the context and the project. In both cases, we have not been able to know which previous commit could be the cause of the bug, so we classified them as *Undecided*.

[6]http://activity.openstack.org/dash/browser/
[7]https://wiki.openstack.org/wiki/Getting_The_Code
[8]https://launchpad.net/openstack
[9]https://review.openstack.org/
[10]One review was discarded as it was left empty.

	Bug Report	Not Bug Report	Undecided	Total
R1	(184) 55%	(115) 34%	(35) 11%	334 (100%)
R2	(188) 76%	(54) 22%	(7) 3%	249 (100%)
R3	(188) 56%	(116) 35%	(30) 9%	334 (100%)
Agree	(209) 72%	(74) 25%	(9) 3%	292 (100%)

Table 2: Statistics for each researcher as a result of the classification process. For each researcher R, the number of tickets (and percentages) classified into the three groups is given. The *Agree* row gives the number of tickets (and percentages) where two researchers agreed.

	Nova	Cinder	Horizon	Neutron	Total
R1 – R2	(44) 70%	(40) 77%	(37) 60%	-	(121) 68%
R1 – R3	-	(46) 73%	(48) 76%	(26) 62%	(120) 71%
R2 – R3	(41) 66%	(10) 100%	-	-	(51) 71%

Table 3: Concordance among researchers for each repository.

	After Deleting Noise
Cause	(179) 51%
Not Cause	(129) 37%
Undecided	(41) 12%

Table 4: Number of times (and percentage) where the previous commit is the cause, not the cause or could not be classified, after deleting noise.

If we attend to how many previous commits each of the 209 bug reports analyzed had, we see that 146 only had a previous commit as in Figure 3, whereas 63 had more than one previous commit as the one in Figure 7. In Table 5, from the 146 unique previous commits, 77 were the cause of the bug, while 30 did not cause the failure and 39 we didn't know the cause. For the 63 bug reports that had more than one previous commit, a total number of 203 previous commits were identified; of them 102 were the cause of the bug, while 99 were not and 2 we were unable to decide.

	One previous commit	More than one previous commit
Cause	(65) 50%	(86) 48%
Not cause	(30) 23%	(82) 46%
Undecided	(36) 27%	(11) 6%

Table 5: Probability of being the cause of a bug depending on if just one previous commit or more than one previous commits are identified.

We also studied the distribution of the number of previous commits for each bug. This result will provide further insight into the bug-seeding nature; it offers as well an idea of the complexity of identifying the cause of a bug, as the more commits involved, the harder it is to identify the cause and understand it. As shown in Table 6, usually the number of commits that can be considered as previous is 1 (over 69% of the cases), followed by 2 commits (13%). In around 17% of the cases, 3 or more commits are involved.

Finally, we were interested in analyzing, for those cases where more than one previous commit exist, how many of them introduced the bug in the code source. Even if several previous commits are involved, it may be the case that none, at least one of them or all of them is the cause of the bug.

Results are given in Table 7; in 16 bug reports all the previous commits were identified as the cause, in 33 bug reports at least one of the previous commits caused the bug, and in 12 bug reports none of the previous commits introduced the bug. If we look at bugs that had two previous commits, in 9 cases both commits were the cause, in 13 cases only one of them was the cause and in another 5 cases non of them could be determined as the cause.

RQ2: Only 50% of the previous commits analyzed caused the failure in the system, whereas the 37% of them did not introduce the bug in the code source.

6. DISCUSSION

The identification of an issue as a bug report is a process that is not as straightforward as one might think. Out of 459 tickets we were only capable to achieve a consensus for 292 cases (63.6%), which hints to the difficulty of the task. Tom help the humans taking the decisions, we investing in building a tool to assist them, by presenting all the relevant information, coming from different data sources, in an easy way. That helped to make the process smoother, but still the differences in judgment, with the same information available, persisted.

After analyzing several hundreds of bugs in this study, we have realized as well that determining where and when a bug was introduced in not a trivial task. In fact, even just determining if the bug was present at the time a certain commit introduced the code that latter triggered it is in some cases not easy.

For example, we have found many cases where we have been unable to determine the point of introduction of the bug, as no previous commit can be identified. One of those cases is when the fix only adds code: in such case there is no way of identifying the previous commit as we defined it, since there is no previous commit touching the fixing lines. In this case, only further research could find out if the code surrounding the added lines was responsible for the bug, or maybe some other part of the code.

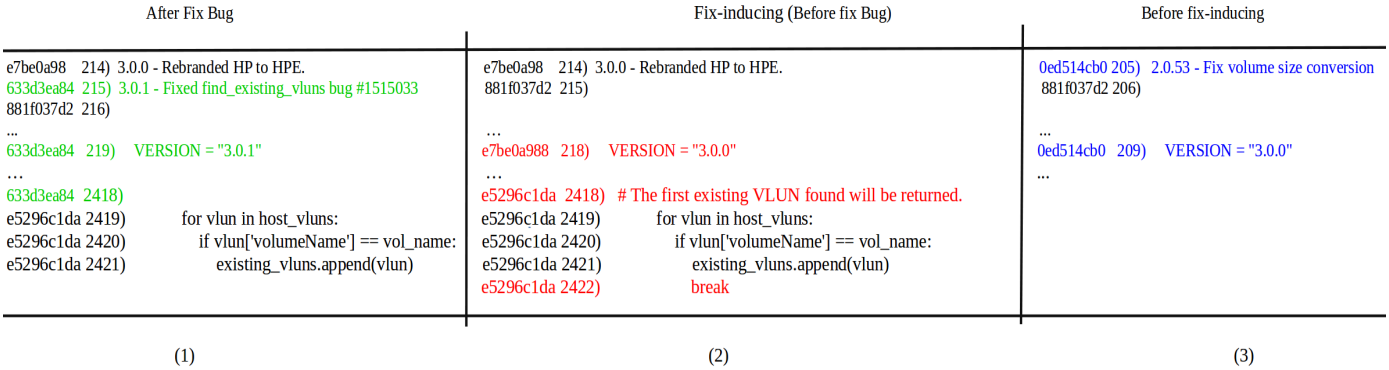

Figure 7: Example of previous commit where more than one commit has been identified. On the left in subfigure (1), the file after the bug-fixing commit. Previous commit *e5296c1da* (subfigure (2)) is the cause of the bug in line 2422 with the presence of an incorrect `break` statement. Previous commits *e7be0a988* (subfigure (3)) has also been identified; however, as it just changes the version number (line 218), it can be classified as not causing the bug.

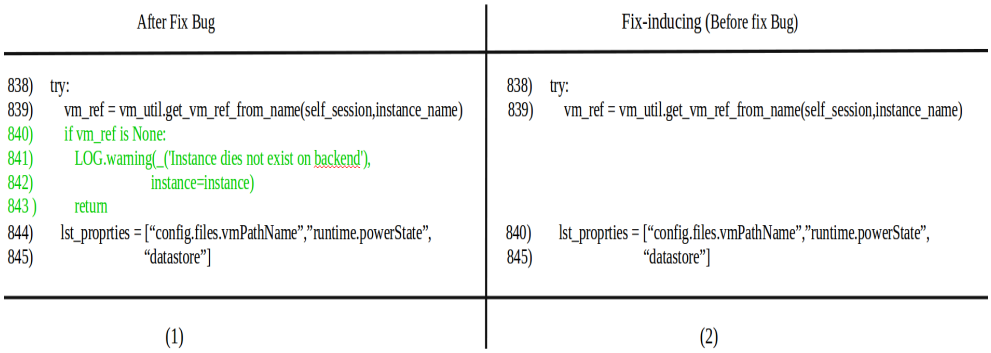

Figure 8: Example of scenario where the bug fix is composed only of lines added. The fix bug commit with lines added in green can be seen on the left (subfigure (1)), and the file as it was before the fix bug commit on the right (subfigure (2)).

Another case is when additional conditions are added to *if* or *else* clauses. One might think that in the previous commit those were not included due to an error, so that the previous commit is the cause of the bug. However, situations exist where the need for additional code in an *if* clause appear because of the later introduction of some new functionality, and thus the line in the previous commit was correct at the time it was introduced. In our analysis, if the latter situation is not explicitly mentioned we have considered that the previous commit caused the error.

In studies such as Kim et al. [16] and Williams et al. [23] also manually analyzed samples of SZZ data. However, they report much higher percentages of correct SZZ results than in this study. The reason could be linked to the time spent in the manual verification (40 seconds in mean), which was lower than us, so they maybe didn't entered into details as much as we have did. Whereas we have analyzed the whole context of a bug-fix they only focused in verify the bug-fix hunks, thus they could omit changes in the API or changes due to the evolution of the code. In the other hand, Williams et al. [23] only took 25 random bug fix commits to manually analyze, and probably it was a small size population. Of course, it could also be because of the specific projects selected in the studies, which could lead to very different cases of bug introduction by previous commits. This is one of the reasons we propose further work to extend our analysis to a varied set of projects.

In any case, our research shows evidence that assuming that the previous commit is where the cause of a bug can be found does not hold for a significant fraction of bugs. The most common reasons for the previous commit not introducing the bug are:

- Changes to APIs, such as the addition of an argument.
- Variable renaming.
- Changes in the operating system, packages or requirements.
- Optimization of the code in some lines.
- Grammar errors, dragged from former commits.

7. THREATS TO VALIDITY

The main threat to validity is the number of tickets considered. It is relatively high, but there is a long way to get a representative sample from a variety of free/open source systems, or software projects at large. Our analysis requires

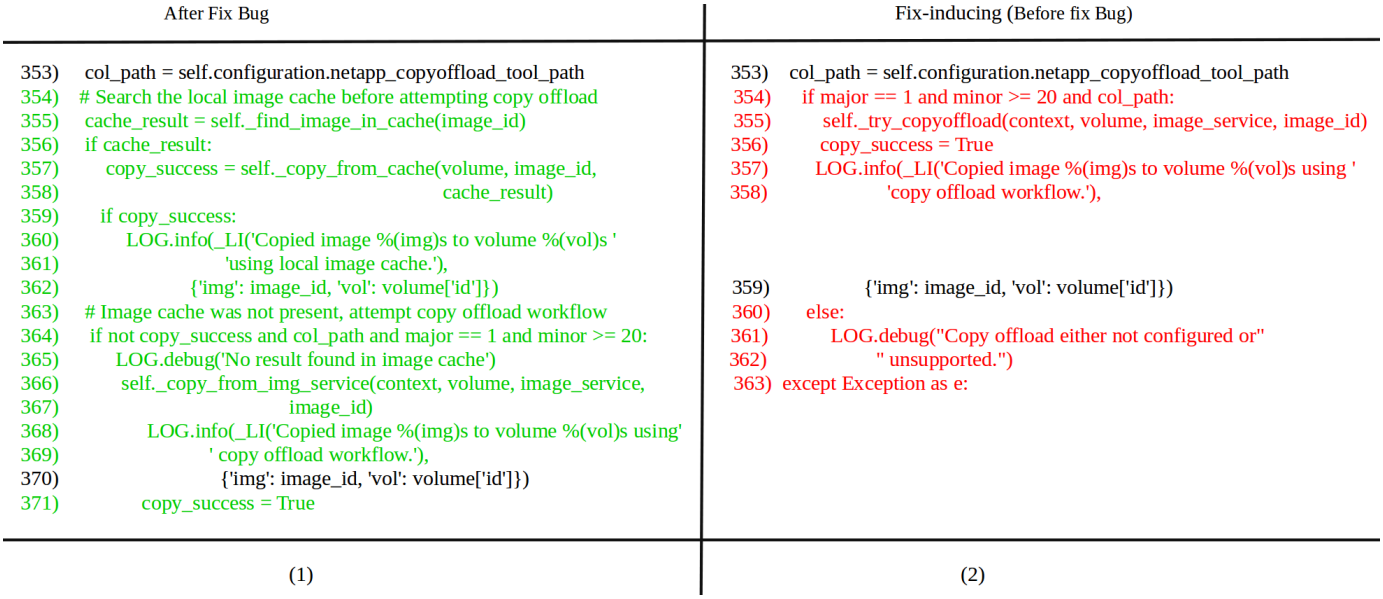

Figure 9: Example of previous commit where we have been unable to decide which line contains the bug. The left code snippet contains the code after the bug fix; the right code snippet shows the code before the bug fix.

	One previous commit	two previous commits	three previous commits	four previous commits	+five previous commits
Neutron	13	5	2	0	1
Horizon	38	9	3	0	4
Nova	48	8	2	4	5
Cinder	47	6	8	3	3
Total	146	28	15	7	13

Table 6: Distribution of the number of commits that can be considered as the previous commit per bug report for each project.

a lot of human effort, so increasing meaningfully the number of tickets is difficult. However, it should be noted that our numbers are the order of magnitude of similar studies: for instance Hindle's et al. article on large commits, considered 100 [11] commits.

Other internal threats to validity are:

- We have not considered those tickets where the two researchers showed discordance.
- We have not taken into account errors that have been classified into *Undecided*, and probably we have lost some *actual* bug reports.
- There could be some lax criteria involving the subjective opinion of the researchers.
- Although the researchers are experienced programmers, they are not experts in the OpenStack project, and their inexperience may have influenced the results of the analysis.
- We are only using part of the information that the tickets provide, like comments and text. There could be some patterns that can be found in other parts of the information.
- We have used a random script to extract the tickets from Launchpad that have been reported during 2015. There could be unintended bias of the data, because many reasons, as for instance the phase of the project.
- In some cases, researchers may have classified the previous commit as the cause of the bug, even if this may not be the case (see discussion on additional conditions in if statements).

The most important external threats, most of them related to peculiarities of the OpenStack project, are:

- The word *bug* is continuously mentioned in the description of commits and tickets, even when it is not an error. This could lead to the incorrect classification during the reviewing process.
- Some tickets are not explicitly described, which could increase the percentage of *Undecided*. This is especially true if the reviewers are not from OpenStack.

	two previous commits	three previous commits	four previous commits	+four previous commits	Total
One are the cause	13	1	1	4	19
Two are the cause	9	7	3	1	20
Three are the cause	-	4	0	2	6
Four are the cause	-	-	0	1	1
+Four are the cause	-	-	-	3	3
None is the cause	5	3	3	1	12
Undecided	1	0	1	0	2

Table 7: Number of previous commits identified as the cause of a bug per bug report

- OpenStack is a special project with a very rapid evolution, and a very active community of developers. Maybe, in other projects with less commits per year, results may be totally different.

8. CONCLUSIONS AND FUTURE WORK

The empirical experiment we have performed in OpenStack has shown that the previous change in the lines touched to fix a bug didn't introduce it for a large fraction of the analyzed bugs: about 37% of the previous commits were not the cause of the bug.

In many cases, we have identified which ones are the changes that actually introduced the bug, which could be useful to improve the accuracy of tools and models developed to prevent bugs. Also, software developers can benefit from identifying where the bug was inserted, improving their processes.

Once we have found that at least in OpenStack the previous commit is in many cases not responsible for introducing a bug, it makes sense to explore, as future work, to which extent this happens in other projects. In particular, OpenStack is a project written mostly in Python, and very active, adding new functionality at a high pace. From the language point of view, it would be interesting studying projects less flexible about types and parameters in API calls, since changes in the API could be in those cases caught by the compiler, and don't become bugs. This is important because in OpenStack many cases of bugs not introduced by the previous commits were because of changes in APIs. With respect to activity, continuous changes in functionality may cause that many bugs are introduced by changes to the code that are not reflected in other parts that use it, leading to most cases of bugs not introduced by the previous commit.

Of course, another future line is the classification, in different projects, of all the cases of introduction of bugs that are not related to the previous commit. This could help to better designing integration tests, to better check for those cases.

The full automation of the methodology used in this paper is also interesting from a practical point of view. That would provide software projects with a valuable tool for understanding how they are introducing bugs, and therefore design measures for mitigation.

Replication package: we have set up a replication package[11] including data sources, intermediate data and scripts.

9. ACKNOWLEDGMENTS

We thank Dorealda Dalipaj and Nelson Sekitoleko, two PhD students in our research team, for their participation in the process of classifying bug reports. We also want to express our gratitude to Bitergia[12] for the OpenStack database and the support they have provided when questions have arisen. Finally, we acknowledge the Spanish Government, because all authors are funded in part by it, through project TIN2014-59400-R.

[11] http://gemarodri.github.io/2016-ICSME-prevcommit/

[12] http://bitergia.com/

10. REFERENCES

[1] E. I. Altman. Financial ratios, discriminant analysis and the prediction of corporate bankruptcy. *The journal of finance*, 23(4):589–609, 1968.

[2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM, 2008.

[3] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider. Bug introducing changes: A case study with android. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 116–119. IEEE Press, 2012.

[4] D. Čubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 408–418. IEEE, 2003.

[5] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162. ACM, 2011.

[6] M. Fejzer, M. Wojtyna, M. Burzańska, P. Wiśniewski, and K. Stencel. Supporting code review by automatic detection of potentially buggy changes. In *Beyond Databases, Architectures and Structures*, pages 473–482. Springer, 2015.

[7] M. Fischer, M. Pinzger, and H. Gall. *Analyzing and relating bug report data for feature tracking*. IEEE, 2003.

[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.

[9] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394–1408, 2009.

[10] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[11] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

[12] D. Izquierdo-Cortazar, A. Capiluppi, and J. M. Gonzalez-Barahona. Are developers fixing their own bugs?: Tracing bug-fixing and bug-seeding committers. *International Journal of Open Source Software and Processes (IJOSSP)*, 3(2):23–42, 2011.

[13] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[14] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on*, 39(6):757–773, 2013.

[15] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[16] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[17] K. A. Neuendorf. *The content analysis guidebook*. Sage, 2002.

[18] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[19] L. Prechelt and A. Pepper. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. *Information and Software Technology*, 56(10):1377–1389, 2014.

[20] V. S. Sinha, S. Sinha, and S. Rao. Buginnings: identifying the origins of a bug. In *Proceedings of the 3rd India software engineering conference*, pages 3–12. ACM, 2010.

[21] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *Proceedings of the 2005 International Workshop on Mining software repositories*, pages 1–5, 2005.

[22] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 812–823. IEEE, 2015.

[23] C. Tantithamthavorn, R. Teekavanich, A. Ihara, and K.-i. Matsumoto. Mining a change history to quickly identify bug locations: A case study of the eclipse project. In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pages 108–113. IEEE, 2013.

[24] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.

[25] H. Yang, C. Wang, Q. Shi, Y. Feng, and Z. Chen. Bug inducing analysis to prevent fault prone bug fixes. In *Proceedings of the Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE 2014)*, pages 620–625, 2014.

[26] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36. ACM, 2011.

[27] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 9–9. IEEE, 2007.