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

It is common practice that developers mark in the versioning system when they are fixing a software bug. At first glance, it could seem reasonable to assume that the fixed bug had been introduced in the previous modification of those same parts of the source code (i.e., in the previous commit). In fact, many studies on bug seeding start with this assumption. However, there is little empirical evidence supporting it, and there are reasons to suppose that in some cases the bug may have been introduced by other actions, such as an older modification, or a change in the API that is being called.

This paper tries to shed some light on this topic by analyzing the relationship of bug fixes with their previous commits. To this end, we conducted an observational study on bug reports, their fixes, and their corresponding previous commits for the OpenStack project. Our results show that the assumption that bugs have been introduced in the previous commit does not hold for a large fraction (at least 37%) of the bugs analyzed.

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 at first reasonable to assume that previous modifications of this line or lines are the cause of the bug. That previous modification is what will refer through this paper as the *previous commit*.

But in fact, to find when and where a bug was introduced in the source code is not a trivial task, and by far more complex than this assumption. This 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 [17], as the authors 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 mining software repositories start with this implicit assumption. As an anecdotal evidence, we have found the following rationales in several areas of research:

- bug seeding studies, e.g., *"This earlier change is the one that caused the later fixed"* [20] or *"The lines affected in the process of fixing a bug are the same one that originated or seeded that bug"* [11],
- bug fix patterns, e.g., *"The version before the bug fix revision is the bug version"* [15],
- 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"* [5].

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 a bug in the source code and understanding the possible causes. For this we took into account when the line was inserted and the general context of the project at that point.

Figure 1 shows a clear example of what we understand as the cause of the 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 written to fix the bug.
2. The code in the middle (subfigure *(2)*) shows the moment in which the bug was introduced (being *31f08423* the id of change), the previous commit.
3. The code on the right (subfigure *(3)*) shows how in previous versions of the file, the bug did not exist.

According to the description in the log of the commit that fixed the bug (see Figure 2), commit *31f08423* was the one where the bug was introduced, as it used a string variable, while a Boolean had to be used, keeping the concordance

MSR '16 Austin, Texas USA

DOI: 10.475/123_4

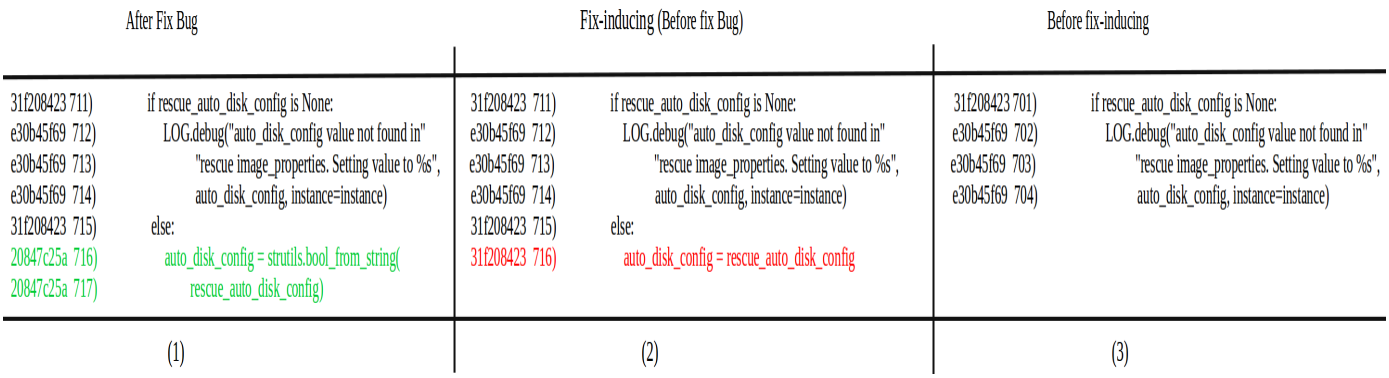

Figure 1: Example of a change in which the bug was introduced in the previous commit. More recent versions of the code are on the left.

```
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.

with the rest of the code. So, in this case, the bug was introduce in the previous commit.

On the other hand, Figure 3 shows a clear example of a case where the cause of the bug cannot be attributed to the previous commit. In this example, the bug fixing commit log (see Figure 4) describes that the name of an argument changed when updating the version causing the failure in the software. This change was done because of the new requirements in the software version, and is unrelated to the changes performed in the previous commit. When the modified lines where introduced the first time, they where not buggy.

```
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.

Based on anecdotal evidence like the one presented in Figures 3 and Figure 4, we argue that in projects that are continuously evolving, with a large development community, code that at some point was correct could become buggy later. Changes in other parts of the code may trigger wrong behavior (bugs) in places which were correct in the past. This happens often in situations like changes of the API. In the moment the code was written, it was correct and the software worked fine. Additions of new features or enhancements to the API may have as a side effect that the formerly correct code starts 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.

In detail, we attempt to address the following research questions:

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

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

But be able 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?*. 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 identify those as a previous step to analyze if they have been caused by the previous commit.

One interesting aspect of our study is that it addresses a very fundamental aspect of many studies on how bugs are fixed: the underlying assumption that there must be a commit previous to the fix, touching the same lines that were later fixed, when somebody introduced the bug. If some 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 when changing it fixes the bug. Therefore, any result obtained after this assumption should be revisited with some

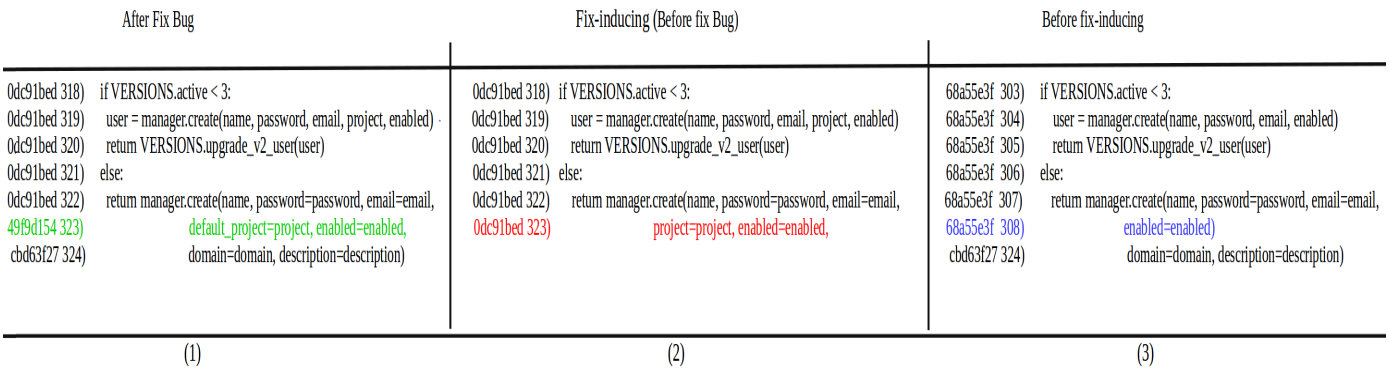

Figure 3: Example of a change where the previous commit, 0dc91bed, did not insert the bug. More recent versions of the code are on the left.

care. We want to contribute with a first step in removing this uncertainty.

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 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

Bug-introducing code changes has been a *hot* topic in the mining software repositories research community for over a decade. The most used algorithm to automatically identify bug-introducing code changes has been proposed by Sliwerski et al. [18], which is an improvement to previous approaches [3, 6, 7]. 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. [14] 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 bug-inducing changes and identify types of changes [20].

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 [18] has been cited, according to Google Scholar, 463 times as of January 2016. An enhanced version of the SZZ algorithm [14] 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. So, 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 [21]. Zimmermann et al. use it for predicting bugs in large software systems [23]. Kim et al. show how to classify file changes as buggy or clean using change information features and source code terms [13]. Kamei et al. apply it to validate effort-aware bug-prediction models [12]. Eyolfson use it to study if time of the day and developer experience affect the probability of a commit to introduce a bug [4]. Izquierdo et al. use the SZZ algorithm to see if developers are fixing their own bugs [11]. Yin et al. use SZZ to find how many fixes to bugs introduce new bugs [22]. Tantithamthavorn et al. employ it to quickly identify the location of a bug [19]. Fejzer et al. use it to support code review [5]. Asaduzzaman et al. apply the SZZ algorithm on Android to study its maintainability [2].

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" [16]. Some methods that consider sources of information other than the previous commit already exist. So, German et al. [8] 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.

Sinha et al. present another technique to identify the origins of a bug in [17]. Their technique is not text-based technique, as 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 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

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

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 mining software repositories research literature in the last years. So, Pan et al. classify the different types of bug-fix patterns that most commonly exist [15]. Antoniol et al. have use a text-based approach to classify bug reports into corrective maintenance and other kinds of activities [1]. 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 [9]. They show that almost 40% of the files marked as defective did not have a bug, and suggest that this misclassification introduces bias in bug prediction models and tools.

3. METHODOLOGY

All data needed to analyze when the bug was introduced can be obtained from the issue tracking systems and the code review systems used generally by free/open source software (FOSS) projects. In our analysis, we have focused on Launchpad[2] as issue tracking system, and Gerrit[3] as code review supporting tool, as they are widely used by FOSS projects nowadays, but our methodology should be adaptable to any such tool.

The Launchpad of each project 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.

3.1 Fist 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[4] 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 next parameters for each ticket:

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

We can see a screenshot of the web interface of the tool in Figure 5. The left side is used to display the information extracted from Launchpad and Gerrit, and the right part is the one in which the 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.

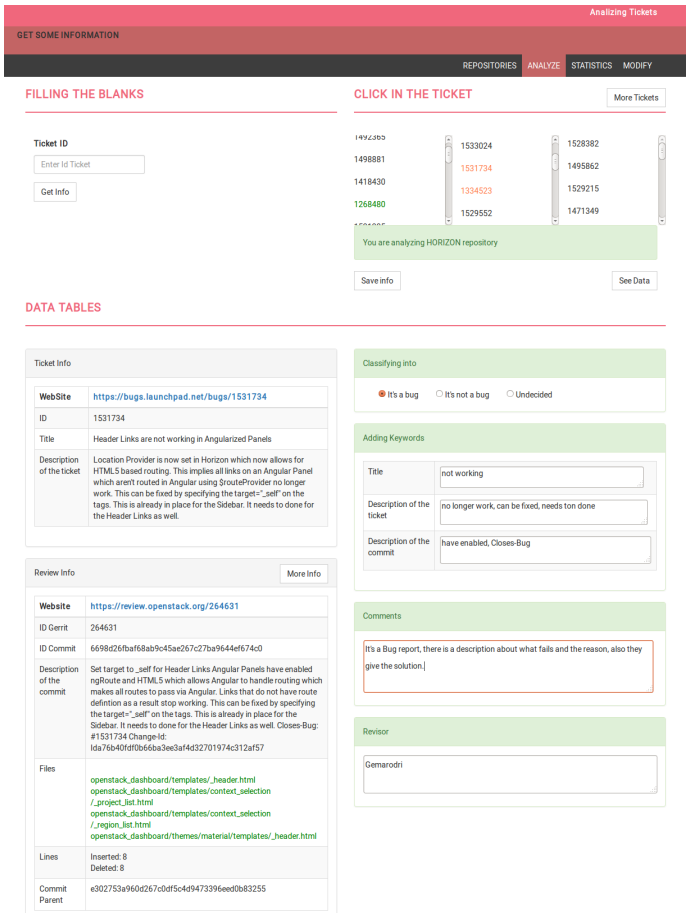

Figure 5: Screenshot of the tool used to classify the tickets.

Each ticket was then categorized into one of three following 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 not bug reports.
3. Group 3 (*Undecided*): The ticket presents a vague description and cannot be classified without doubts.

[2]https://launchpad.net/
[3]https://www.gerritcodereview.com/
[4]bugtracking.libresoft.es

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

1. Each time that the title or the description of a ticket describes an unexpected behavior in the program, our criteria indicated that it was considered as a bug report.
2. 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 were required, 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.
4. When only test files are affected in a ticket, we classified it as not being a bug report. We consider bug errors in test files are a different type of bugs, as the software may still work as expected.

Sometimes we were unable to answer all the questions due to having insufficient data or because of the complexity of the issue. In this case, the ticket was 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 gathering additional information on the context of 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 the correct change, because sometimes although the commit added many lines, if you look the code before the commit you can check that some of the lines added was there, and in that case, it is a false positive where the previous commit did not cause the bug.

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.

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.

2. git checkout *parent of commit that fix 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 inserted 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. So, 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

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[5]. All its history is saved and available in a version control system[6], as well as its issue tracking system (Launchpad[7]) and the source code review system (Gerrit[8]).

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.

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. Each ticket was analyzed by two 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]http://activity.openstack.org/dash/browser/
[6]https://wiki.openstack.org/wiki/Getting_The_Code
[7]https://launchpad.net/openstack
[8]https://review.openstack.org/

	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

5.1 Fist Stage

We classify a total of 459 tickets using the tool, resulting in 917 reviews[9]. Only those tickets classified as bug reports by both researchers were considered in the next stage, which analyzes if the cause of the bug was introduced in their previous commits. This process requires manual inspection by researchers. In the mean, classifying a ticket takes between 5 and 10 minutes, although the amount of time decreases with experience as could be expected.

Table 2 shows the classification percentages of the analized 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 189 tickets[10] of the classified as *Bug Reports*, 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 348 previous commits which could be the cause of the 189 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 or even a change in the version of the file. So, in total we have analyzed 308 previous commits.

As can be seen in Table 4, from the 308 previous commits, 152 have been considered to be the cause of the bug, whereas 114 have been identified as not being the cause.

We have been unable to decide in 42 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*.

	Before Deleting Noise	After Deleting Noise
Cause	(152) 44%	(152) 49%
Not Cause	(154) 44%	(114) 37%
Undecided	(42) 12%	(42) 14%

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

If we attend to how many previous commits each of the 189 bug reports analyzed had, we see that 131 only had a previous commit as in Figure 3, whereas 58 had more than one previous commit as the one in Figure 7. In Table 5, from the 131 unique previous commits, 65 were the cause of the bug, while 30 did not cause the failure. For the 58 bugs that had more than one previous commit, a total number of 179 previous commits were identified; of them 86 were the cause of the bug, while 82 were not.

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 10% 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

[9]One review was discarded as it was left empty.

[10]20 tickets have not been analyzed due to time constraints.

	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.

	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.

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 8 bug reports all the previous commits were identified as the cause, in 30 bug reports at least one of the previous commits caused the bug, and in 11 bug reports none of the previous commits introduced the bug. If we look at bugs that had two previous commits, in 4 cases both commits were the cause, in 9 cases only one of them was the cause and in another 4 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 experience gained in this study with exposure to several hundreds of bugs allows us to state that determining who (or what) introduced a bug is a non-trivial task.

Although at first, as shown in the Figure 1 and Figure 2, one may think that this is an easy task, we have found many examples where we have been unable to determine the cause as no previous commit can be identified. This is, for instance, the case when only code has been added and there is no way to identify the previous commit. In this case, further research could find out if this is not really the addition of a new feature rather than a bug.

There are other cases where it is difficult to determine if the change is the cause. For instance, when additional conditions are added to *if*s or *else*s, one might think that in the previous commit those were not included due to an error, so that the previous commit becomes the cause of the bug. However, situations exist where the additional conditions in an if appear because of the introduction of a 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.

But already 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 complexity of the task. The amount of information, the number of fields and the requirement of human interaction made us invest time in the creation of a tool that assisted us in the process.

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 not insignificant percentage of bugs.

7. THREATS TO VALIDITY

As any other empirical study, this one presents several threats to its validity, external and internal, that have to be considered and taken into account. In order to allow others to study it in detail, replicate it or even build on top of it, we have set up a replication package[11] including data sources, intermediate data and scripts that can be obtained.

The number of the tickets extracted form Launchpad is relatively high, but probably not as high as to state that it can be representative of all free/open source systems, or the software industry. It has not to be forgotten that the analysis requires a lot of human effort, so that achieving large numbers of cases is difficult. However, it should be noted that the amount of tickets in in the order of magnitude of similar studies found in the research literature, for instance Hindle's et al. article on large commits, where 100 [10].

The internal threats related to the researchers that have conducted the study are following:

- We have not considered those tickets where the two researchers showed discordance.

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

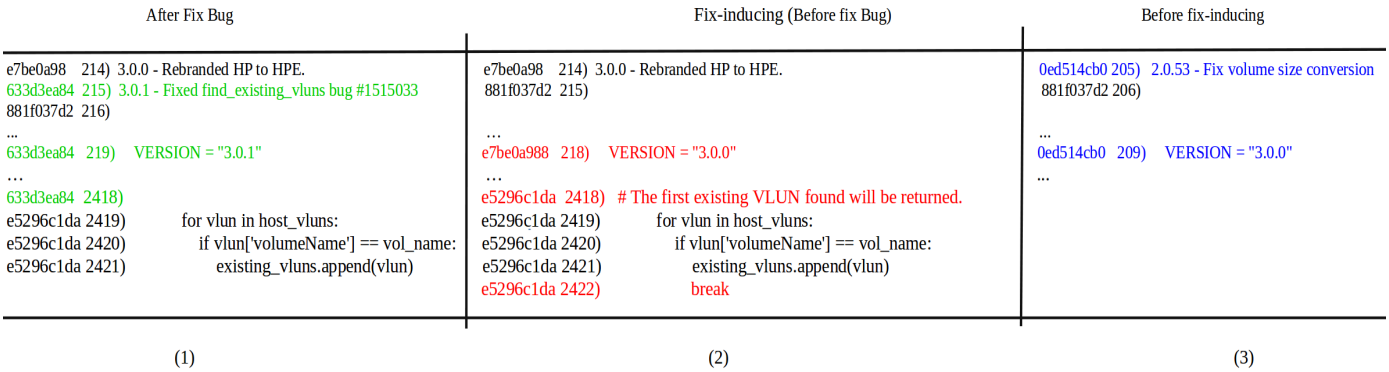

Figure 7: Example of previous commit where more than one commit has been identified. On the left in subfigure (1), the file after the bug-fixing commit. Previous commit *e5296c1da* (subfigure (2)) is the cause of the bug in line 2422 with the presence of an incorrect `break` statement. Previous commits *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.

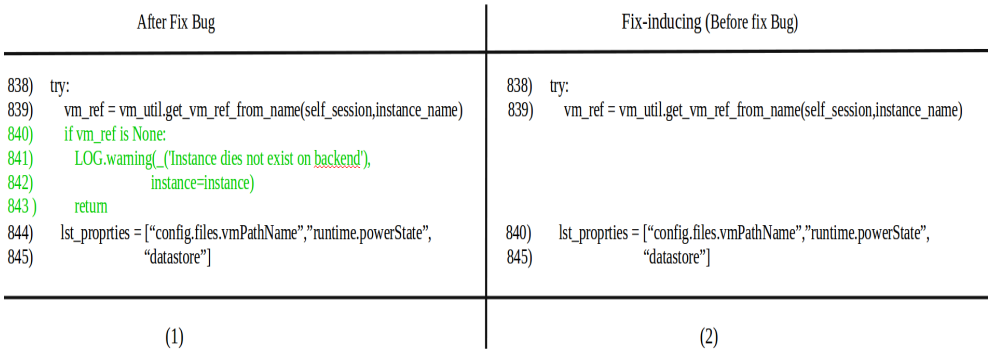

Figure 8: Example of scenario where the bug fix is composed only of lines added. The fix bug commit with lines added in green can be seen on the left (subfigure (1)), and the file as it was before the fix bug commit on the right (subfigure (2)).

- 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 a recognized pattern in the data, unknown at first sight, that involves 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 external threats, related to the case of the project, are following:

- The word *bug* is continuously mentioned in the description and commit of a ticket even when we found 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.
- OpenStack is a special project with a constant evolution due to their 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 carried out in OpenStack supported that the current premise assumed does not hold for a large fraction of the analyzed bugs, because around the 40% of the previous commits were not the cause of the bug.

With our methodology we have identified which are real changes that introduced the bug, and this could be useful

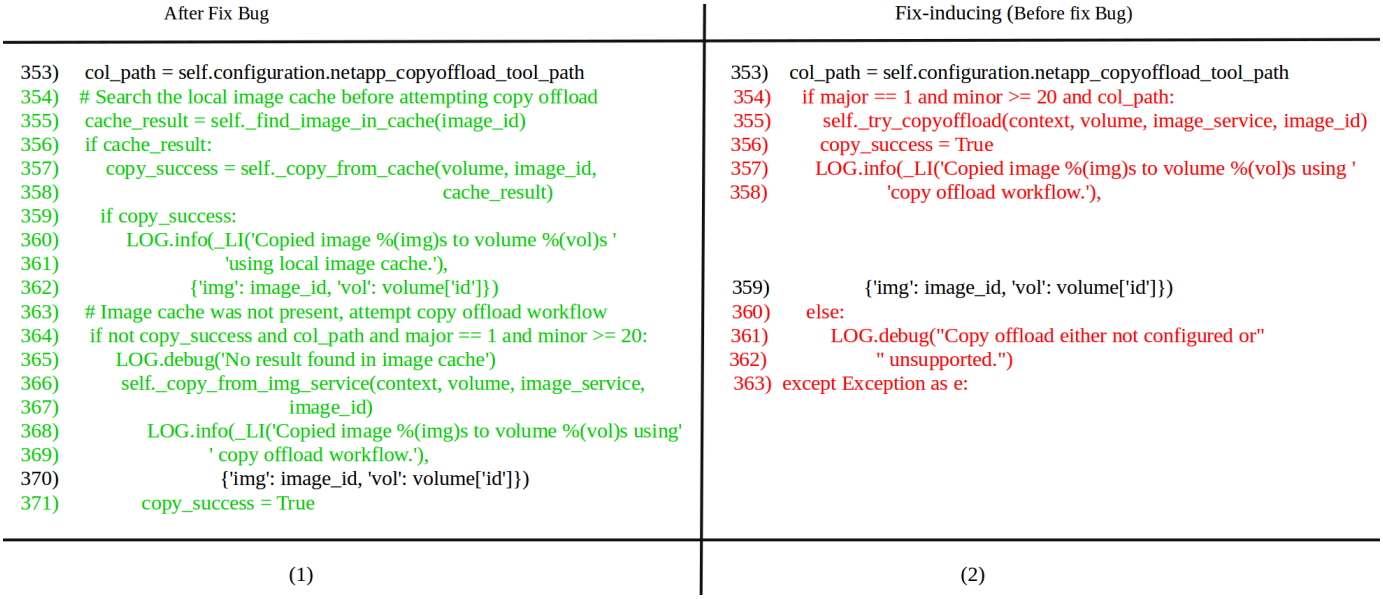

Figure 9: Example of previous commit where we have been unable to decide which line contains the bug. The left code snippet contains the code after the bug fix; the right code snippet shows the code before the bug fix.

	One previous commit	two previous commits	three previous commits	four previous commits	+five previous commits
Neutron	11	3	2	2	0
Horizon	39	8	3	2	4
Nova	44	5	2	4	4
Cinder	37	9	6	2	2
Total	131	25	13	10	10

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

to improve the accuracy of those tools developed to prevent bugs. Also, the software developers stand to benefit from identifying where the bug was inserted, improving their methodology.

A final field of future work could be concerned with the full automation of the methodology presented in this paper, developing an automatic classifier. Another interesting investigation could perform the same empirical study on a project with a less active community, to prove if our idea is fulfill in other projects. So, for projects that do not evolve as rapidly as OpenStack, e.g., those with a more stable API, could offer completely different results even if applying the same analysis methodology used in this paper.

9. ACKNOWLEDGMENTS

We thank Dorealda Dalipaj and Nelson Sekitoleko, two PhD students in our research team, that participated 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 would like to acknowledge the Spanish Government because all authors are funded in part by it, through project TIN2014-59400-R.

[12]http://bitergia.com/

10. REFERENCES

[1] 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.

[2] 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.

[3] 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.

	two previous commits	three previous commits	four previous commits	+five previous commits	Total
All are the cause	4	3	0	1	8
At least one is the cause	9	7	5	9	30
None is the cause	4	2	4	1	11
Undecided	1	0	1	0	2

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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*, page 9. IEEE, 2007.