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

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

Jesus M. González-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 this assumption, 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 of the bugs analyzed. FIXME: maybe something more here

Keywords

Bug introduction, bug seeding, SZZ algorithm, previous commit

1. INTRODUCTION

When a failure is found in the behavior of a software, the developers try to fix it locating and modifying the source code line(s) that are reponsible for the wrong behaviour in the source code. It seems reasonable to assume that the previous modification of this line or these lines are the cause of the bug; this previous modification is *the previous commit*. But in fact, to find when and where a bug was introduced in the source code is not a trivial task. This has largely been ignored, mainly because the data related to the origin of a bug is embedded in the evolution of the software [15].

MSR '16 Austin, Texas USA

DOI: 10.475/123_4

FIXME: clarificar la frase anterior... no ha quedado muy claro.

This is the reason why many studies in the area of mining software repositories start with this implicit assumption. So, for instance, we have found following reasonings in several areas of research, such as:

- in bug seeding studies, e.g., *"This earlier change is the one that caused the later fixed"* [18] or *"The lines affected in the process of fixing a bug are the same one that originated or seeded that bug"* [6],
- in bug fix patterns, e.g., *"The version before the bug fix revision is the bug version"* [13],
- and in tools that prevent for future bugs, e.g., *"We assume that a change/commit is buggy if its modifications has been later altered by a bug-fix commit"* [2].

But although the assumption can be found frequently in the research literature, in our opinion there is not empirical evidence supporting it. That is the reason why we have conducted an observational study on fix-bugs, devoting a significant effort to locate the origin of a bug in the source code and understanding the possible causes. For this we had to take into account the moment in which the line was inserted and the general context of the project.

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)*) ensures that 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 variable as string when a Boolean had to be used, keeping the concordance 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 is not to be attributed to

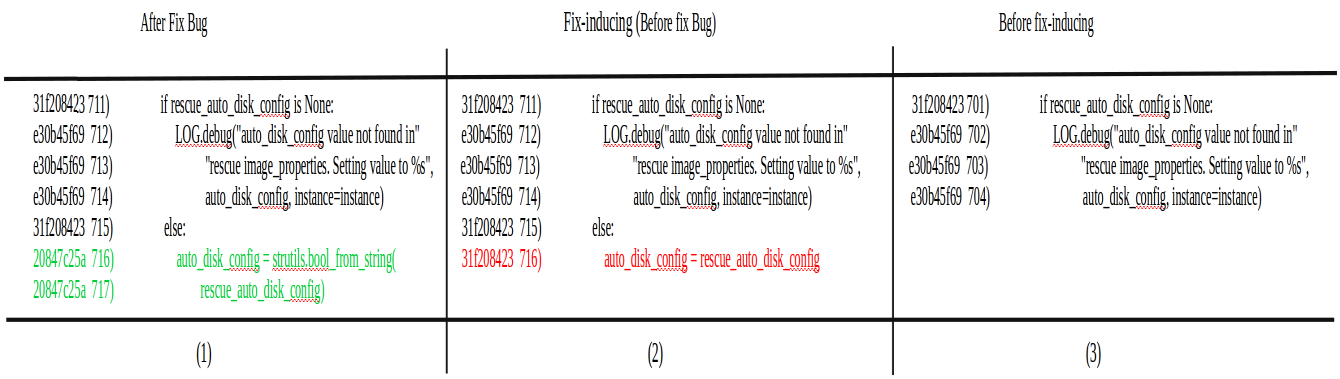

Figure 1: Example of a change in which the bug was introduced in the previous commit. More recent versions of the code are on the left.

```
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.

the previous commit. In this example, the bug fix commit log (see Figure 5) 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 in the first time, they did not contain the bug.

Based on anecdotal evidence like the one presented in Figures 3 and Figure 5, we argue that in projects that are continuously evolving, with an ample developer community, code that before was correct could be buggy at some time. So, changes in other parts of the code may induce wrong behavior (bugs) in places that were correct in the past. This happens often in situations like changes to the API. In the moment the code ws written, it was correct and the woftware worked fine. Additions of new features or enhancements to the API had as a side effect that the formerly correct code presents a wrong behavior, making the software fail. But in such cases, the source of the error cannot be FIXME assigned to a change performed in the previous commit, as in that moment it referred to a different API.

FIXME: poner esto bien

The goal of this paper is to find if the cause of the bug can be FIXME assigned to the previous commit, understanding that at the time when the previous commit is introduced

not all times the code analyzed was so easy as the code showed in figures 1 and 3. So the principal aim of this paper to find the responsible of the bug is to know if the previous commits contained buggy code at the moment in which were added/modified or, another change in the software, related to the update of the software and its evolution, caused the bug.

In detail, in this paper we attempt to address the following research question regarding who introduced the bug in the source code:

FIXME: rephrase RQs

- RQ1: How can we know that a change was done to fix a bug in the source code? How can we identify them?
- RQ2: When did the previous commit introduce the line with the bug into the source code?

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

The first algorithm to identifying bug-introducing code changes automatically was proposed by Sliwersky et al. [16]. Currently, it is a well-known algorithm called SZZ, which is based on text differences to discover modified, added and deleted lines between the bug-fix and its previous version. The SZZ algorightm uses the CVS `annotate` command[1] to identify the last commit that touched these lines.

An improvement to the SZZ algorithm is described by Kim et al. [8]. There the authors used 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.

Sinha et al. present another technique to identify the origins of a bug in [15]. 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.

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

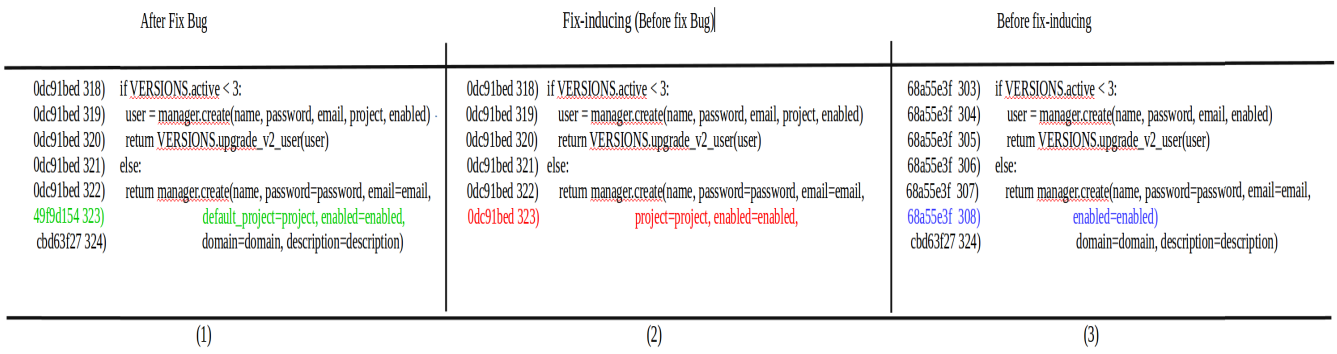

Figure 3: Example of a change where the previous commit, 0dc91bed, did not insert the bug. More recent versions of the code are on the left.

```
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 two approaches have some metodological 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.

FIXME: This last paragraph should be clarified, and more on papers using the SZZ algorithm! Maybe we should state how many papers in total exist based or using SZZ in Google Scholar.

The SZZ algorithm (and its *successors*) have been widely used in the researc community. Williams et al. revisited the SZZ algorithm to track bug-inducing changes and identify types of changes [18]. Yang et al. applied SZZ to find what kind of bug-inducing changes are likely to become a great threat after being mared as bug-fix changes Finally, some bug prediction algorithms are based on SZZ; Kim et al. showed how to classify file changes as buggy or clean using change information features and sour code terms [7].

FIXME: maybe talk about my paper with dmg and ahmed, where bugs could be found elsewhere [3]. There the talk is precisely about those bugs whose origina are elsewhere. Title of the paper: *Change impact graphs: Determining the impact of prior codechanges*

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 generalizable 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 labour 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

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

- The changes to the source code, as sometimes neither the descriptions nor the comments by developers in the Launchpad an Gerrit of each ticket, clarified the underlying ticket. FIXME: what comments?

FIXME: aqui podemos poner un pantallazo de la interfaz web. We can see the web interface of the tool in the image **??**, the left side is used to display the information extracted from Launchapad and Gerrit and the right part is the one in which the users can wirte and classify the ticket into one of the three groups.

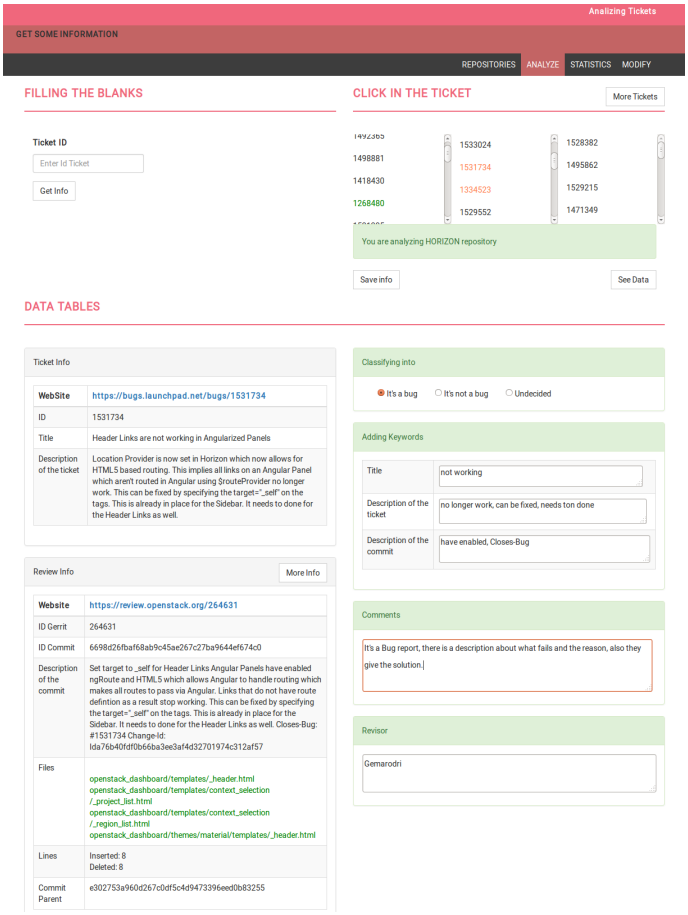

Figure 5: Screenshot of the tool used to classify the tickets

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.

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

1. When the ticket title described the program as not working as expected, our criteria indicated that it was a bug report.

2. When the title described 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 title described that 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. Test files in a ticket were not be analyzed, as we consider they are not a core part of the software and are used as a testing method to determine whether the code is fit for use. When the bug exclusively lies in a test file, the ticket was not considered as bug report because the software still works as expected, only test fails. FIXME: no se repiten las frases?

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?

The next part is focused on analyzing the previous commit exclusively in the *Bug Report* group to identify which line contained the bug and why the software failed, keeping in mind 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. This way, we can be sure that we are looking the correct change, because some times 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, is a false positive where the previous commit didn't cause the bug. FIXME: aclarar este parrafo!

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 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 need to discard some noise presents in our final results according to the responsibility of the previous commit inserting the bug in the code source. Due to they were not responsible for cause the bug, we delete those previous commit which presents the following criteria:

- Blank lines
- Format changes
- Copied lines
- Changes in the comment.
- Updates in the version of a file.

4. EVALUATION

We validate our methodology analyzing 459 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 of its 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, as well as its issue tracking system (Launchpad[6]) and the source code review system (Gerrit[7]).

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

	All History	Last Year (2015)
Nova	14,558	3,283
Fuel	9,139	5,123
Netron	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

In this four projects we analyzed the relationship of bug fixes with their previous commits. In order to identify the moment in which the bug was injected into the source code. The first stage we did automatically using the tool and a double bind between three researchers, three PhD student included me. The second stage was done manually and only me was involved analyzing this relationship.

5. RESULTS

We extracted a total of 459 different tickets from the Launchpad of the four principal projects in OpenStack, 125 tickets from Nova, 125 tickets from cinder, 125 tickets from Horizon and 84 tickets from Neutron. These tickets first, were analyzed to identify those ones which were real bug reports. And secondly, each file was analyzed to obtain which previous commit inserted the bug causing the failure of the system and reporting the bug report.

[5]http://activity.openstack.org/dash/browser/
[6]https://launchpad.net/openstack
[7]https://review.openstack.org/

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

Table 2: Statistics of each researcher in the classification

5.1 Fist Stage

We classify a total of 459 tickets using the tool. In 417 of the tickets, we used double bind analysis, and only those tickets classified as bug report by two of us, were considered in the next stage to analyze the relevance of their previous commits.

The table 2 shows the percentage of each researcher after analyzing the tickets, and the number of tickets classified identically by two different researchers. Obtaining that the researchers R1 and R2 had a similar data in their results, whereas research R2 got results significantly different with a higher number of tickets classified as Bug Report.

Finally, the researchers identified in the same way 292 tickets, that is, their results matched in a 70% of the cases. Obtaining 209 tickets classified as Bug report, 74 tickets classified as Not Bug Report and 9 tickets classified as Undecided.

Also we had measured the concordance in the classification of each developer according to the project analyzed. The table 3 shows that the concordant got between all the three research was very similar, around a 70%. Furthermore, the concordance form each researcher with the rest always was up to 60%.

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

Table 3: Concordance between each developer in each repository

5.2 Second Stage

At this stage, we analyzed the 209 we got a list with all the previous commits and we were be able to classified the previous commit/s as Responsible, Not Responsible or Undecided, taken into account that the bug could be inserted in different lines of different commits, but not everyone had to be responsible for the commit, sometimes the previous commit copied lines from its previous commit or inserted comments and blank spaces. In fact, the responsible can be only one of them, more than one or maybe none.

We identified a total of 348 previous commits which can be responsible for inserting the line containing the bug. After analyzing the Bug Reports and their previous commits and discarded 40 of them because were noise, Table 4, we got

that 152 previous commit were responsible to cause the bug whereas 114 previous commit had not any responsibility in the failure of the system, and only in 42 previous commits we were unable to identify the cause of the bug.

	Before Deleting Noise	After Deleting Noise
Responsible	152	152
Not responsible	154	114
Undecided	42	42

Table 4: Responsibility of each previous commit before and after deleting the noise in the results

Furthermore, focusing on how many previous commits presented each Bug Report, we obtained that 131 had one previous commit implicated, whereas 58 had more than one previous commit implicated in their file/s. According to Table 5, from the 131, we obtained that 65 of them inserted the bug, but 30 of them were not responsible in the failure of the system. And from the 58 which had more than one previous commits, we obtained in total 189 previous commit, where 86 of them were responsible and 82 were not responsible.

	One previous commit	More than one previous commit
Responsible	65	86
Not responsible	30	82
Undecided	36	11

Table 5: Probability of cause the bug depending on how many previous commits had the bug report

Also, we looked the distribution of the previous commit in each Bug Report, Table 6, we observed that the most common distribution in the relationship between previous commit and bug report is one previous commit per Bug Report, following by the second common distribution, two previous commit per Bug Report.

Finally we wanted to know the responsibility practiced by each previous commit in the failure of a system, in other words, we were interested in analyze from those cases where exists more than one previous commit, how many of them inserted the bug in the code source, Table 7. We obtained that in 8 Bug Reports all the previous commits were responsible, in 30 Bug reports at least one of their previous commit caused the bug and in 11 bug report none of their previous commits inserted the bug.

6. DISCUSSION

RQ1: Using all the information available in the bug tracking system and code review system related a fix-bug, we have obtained that this fix-bug were real Bug Reports in a 72% of the tickets analyzed

RQ2: We have confirmed that 50% of the previous commits analyzed caused the failure in the system, whereas the 37% of them didn't injected any bug in the code source

1. No todos los casos son tan claros como los mostrados en los ejemplos ....
2. Casuistica del common juidment
3. Hemos sido conservadores
4. Hemos utilizado la herramienta porque es un proceso complicado

Once we have all the tickets analyzed by diferents researchers who have used a double blind, how to proceed if there are discordances between them:

1. Should they discuss after their analysis to reach a better classification?, Should the tool provide this?
2. Does the Bug report only the same ticket classified as Bug report for all the researchers?

How to proceed if looking for the responsibility of a bug when only added lines are inserted? And we are talking about a bug report not a new feature, these kinds of cases use to be when a researcher forgot check some case inside a function. [reference]

1. Is responsible the function where these lines are content?
2. Is responsible the last commit that modify something in the function?

7. THREATS TO VALIDITY

The size of the tickets extracted form the Launchpad is medium, but doing the analysis manually we are sure that the results present in this paper are valid, being sure that the previous commits classified as not responsible, they are.

Although, we understand that the model presented has some threats, external and internal, that make our model not 100% valid. The internal threats related to the researchers that have conducted the study are following:

- We have not taken into account errors that have been classified into *Undecided*, and probably we are lost some real bug reports belonging this group .
- There could be some lax criteria involving the subjective opinion of the researchers.
- The researchers are not experts in OpenStack, and our inexperience may have influenced the results of the analysis.
- We are only using part of the information that the ticket provides, like comments and text. There could be a recognized pattern in the data, unknown at first sight, that involves other parts of the information.
- Although we use a random script to extract the tickets reported in during the last year, 2015, from the launchpad, in this year could be some bias unidentified.
- In case of the researchers didn't find the information to know if the previous commit inserted the bug or in contrast, it was caused by the evolution of the software. They keep the traditionalist thought, classify these previous commits are responsible.

	One previous commit	two previous commit	three previous commit	four previous commit	+five previous commit
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 number of previous commit per Bug Report in each project

	two previous commit	three previous commit	four previous commit	+five previous commit	Total
All Responsible	4	3	0	1	8
At least one responsible	9	7	5	9	30
None Responsible	4	2	4	1	11
Undecided	1	0	1	0	2

Table 7: Probability of cause the bug depending on how many previous commits had the bug report

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 put down a constant evolution due to their active community of developers. Maybe, in other projects with less commits per year the statistics about the responsibility of previous commit change.

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 responsible inserting the bug. With our results we can identify which ones are real changes that introduced the bug, and this could be useful 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, improve their methodology.

A final field in our future work concerns the full automation on the methodology could developer an automatic classifier base on the idea that not all the previous commit injected the bug. Another interesting investigation could perform the same empirical study in a project with a community less active, to can prove if our idea is fulfill in other projects.

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[8] for the OpenStack database and the support they have provided when questions have arised. Finally, we would like to acknowledge the Spanish Government because all authors are funded in part by it, through project TIN2014-59400-R.

10. REFERENCES

[1] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[8] http://bitergia.com/

[7] 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.

[8] 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.

[9] S. Koch. *Free/open source software development*. Igi Global, 2005.

[10] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[11] E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[12] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 63. ACM, 2012.

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

[14] 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.

[15] 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.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[17] E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1):100–118, 1985.

[18] 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.

[19] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25. ACM, 2011.

[20] 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.

[21] 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.

[22] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr. Mining version archives for co-changed lines. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75. ACM, 2006.

[23] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.