Analyzing how the bugs are injected into the source code.

Gema Rodríguez-Pérez[1]

gerope@libresoft.es, LibreSoft, Universidad Rey Juan Carlos

Summary. There is an ample research in the software engineering literature on software defects. In the field of mining software repositories, it's very important understand how the bugs are injected into the source code to prevent the system to fail. In the currently literature many studies on bug seeding start with an implicit assumption: the bug fixed had been introduced in the previous modification (i.e., in the previous commit) of those same lines of the source code. However, we have conducted and observational study that proved the assumption that bugs have been introduced in the previous commit and the results, showed this assumption does not hold for a large fraction of the bugs analyzed.

Our objetive is shed some light on bug seeding topic by analyzing how the bug are inserted into the source code and understanding why the bug appears in the source whereas the developers are at their disposal code reviews and automatic inspections. We pretend conducted an large observational study that involved bug notifications from a free and open-source cloud computing software platform in order to find some pattern that can help us preventing the bugs.

Key words: Bug introduction, bug seeding, SZZ algorithm, previous commit

1 Introduction

Many efforts on how and why bugs are introduced in the software source code are underway in the software engineering research community. Software source code is affected by many changes, many of them due to failure of the software because of emergent bugs. Developers try to fix them by locating and modifying the line(s) of source code in which the bug is. Concepts such as *bug seeding* help us to find how and where a bug was inserted in the source code, and should be reasonable to assume that last modification, or *previous commit*, of this line or these lines injected the bug.

In spite of the many studies in the area of mining software repositories that are based on this implicit assumption, it is not a trivial task to find when and where a bug has been introduced in the source code, and thus to identify who introduced the bug. There are some reasons to assume that in some cases the bug may not have been introduced in the previous commit, being other actions such as change in the API that is being called or an older modification the cause for a bug. But in fact, this has been largely ignored in the related work; as anecdotal evidence in papers of different areas of research the following statements can be found:

- in 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"* [9],
- in bug fix patterns, e.g., *"The version before the bug fix revision is the bug version"* [15],
- in 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"* [4].

While performing research on the topic, the unique empirical evidence found in the literature that supports this assumption is based on a manual verification of 25 random bug-fix commits with some improvements in the use of the SZZ algorithm, concluding that the SZZ intuition in which the change previous to a bug fix introduces the bug is fulfilled [20]. But this empirical evidence is not enough, it only takes a small population of bug-fix commits, and we need more empirical evidence because of the assumption can be found frequently in the literature. That is the reason why we decided to investigate its validity in the case of a large project, such as the OpenStack, pinpointing the origin of a bug in the source code and devoting significant effort to understand the causes.

The many changes to the code in this project enables us to identify bug reports in which the bug had not been introduced in the previous commit. As we have mentioned before, an example of this could be a change in the API that is being called. For instance, the code presented below shows a real code extracted from OpenStack in which a certain volume doesn't work with multiple backend enable, due to in the current design it was not necessary, see red lines in *Bug-Insertion (V2)*, but in a certain moment and for some reasons, the community need it. So, the fixed bug added a new value in API call, see green lines in *Bug-Fix (V5):*.

The Figure 1 shows an example of the history of commits done in a file, we can see the current version, *V6*, the commit that fix the bug, *V5*, the commit that injected the bug according with SZZ intuition, *V2*. The commit *V1* is the first time that the function involved in the bug fix appear in the file, and the commits *V3, V4* are different states of the file.

Before Bug-Insertion (V1):

```
dbb854635 xioaxi 2013-07-11       def _check_backup_service {self,volume}:
afd69a95b victor 2013-08-28       """Check if there us an backup service available."""
dbb854635 xioaxi 2013-07-11       topic = CONF.backup_topic
```

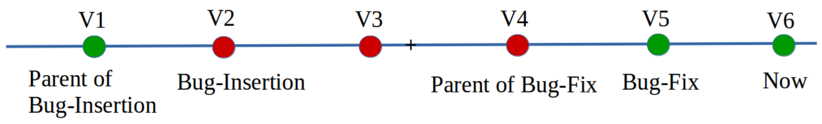

Fig. 1. Control version history with the bug-insertion commit and the bug-fix commit

```
            ...                                ...
dbb854635 xioaxi 2013-07-11            srv['host']==volume['host'] and not srv['disable']
```

Bug-Insertion (V2):

```
d6dd5cdfa keniche 2013-09-05        def _is_backup_service_enabled {self,volume}:
afd69a95b victor 2013-08-28        """Check if there us an backup service available."""
dbb854635 xioaxi 2013-07-11        topic = CONF.backup_topic
            ...                                ...
dbb854635 xioaxi 2013-07-11            srv['host']==volume['host'] and not srv['disable']
```

Bug-Fix (V5):

```
bd5c3f5a0 Edward 2013-09-26        def _is_backup_service_enabled{self,volume,volume_host}:
afd69a95b victor 2013-08-28        """Check if there us an backup service available."""
dbb854635 xioaxi 2013-07-11        topic = CONF.backup_topic
            ...                                ...
bd5c3f5a0 Edward 2013-09-26            srv['host']==volume_host and not srv['disable']
```

In this example, according to SZZ algorithm the previous commits *d6dd5cdfa and dbb854635* inserted the bug; due to some drawbacks in this algorithm, the approach carried on [13] added some improvements removing some false positives. So according with this approach, the commit *dbb854635* was who inserted the bug, because the other commit, *d6dd5cdfa*, only modified the name of the API call and it is a false positive. But this intuition in this example in not fulfilled, because of the two lines involved in the bug-fix which were inserted in the *V1* haven't any bug, this function works fine at these moment.

The goal of my PhD, at this moment, is to find out to which extent the cause of bugs can be attributed to the previous commit, and shed some light to understand why and what the bug appeared. This is the reason why we conducted an observational study that involved issue notifications from the most active components in the OpenStack project. From the issue notifications we have to analyze only those that are bug notifications, so before analyzing who and what caused a bug, we have to be able to identify the bugs reports. We think that this observational study is the first step to identify and understand how, when and why the bug was injected into the source code.

In detail, we would like to answer our main research question, *How the bugs are seeding in the code?*. But firts, we attempt to address the following research questions, some of them are methodological questions:

- RQ1: Which reports in the issue tracking system are (real) bug reports?

- RQ2: How often is the previous commit the cause of the bug?

The results from the above research questions, give cause for attempting address the following research questions in a future study;

- RQ3: Could we find some pattern in the bug seeding?
- RQ4: Are these patterns present in the different programing lenguages?
- RQ5: How could we prevent these bugs ?

The remainder of this paper is structured as follows; first, we present the related work in Section 2, in Section 3 we detail the two stages carried out in our methodology. Results from the two stages are then shown in Section 4. Finally, we present the future work and the possible applications of our findings in Section 5 .

2 Related Work

The most used algorithm to locate automatically bug-introducing code changes by linking information from version control system to a issue tracking system repository has been proposed by Sliwerski et al. [19]. This algorithm is an approach of previous works [2, 5, 6] and it is a well-known algorithm, called SZZ.

The SZZ algorithm has some limitations linking bug fixing commit and to bug-introducing commits, because of it only looks for some special keywords in commit messages (e.g., “Bug” or “Fix” [11]). If a bug-fix commit message does not contain the keywords, this bug-fix commit will be ignored. This is the reason why there are articles that suggest improvements on SZZ algorithm such as the Kim et al. [13], it uses annotation graphs instead of CVS annotation to locate, in the previous versions, the lines involved in a modification or a deletion. Also, they discarding some false positives as blank spaces, changes in the format or changes in the comments.

There are studies based on SZZ, Williams et al. have revisited the SZZ algorithm to track bug-inducing changes and identify types of changes [20]. 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]. Bavota et al. reported an empirical study about what extent refactoring activities induce faults obtaining that a few percentage of bug fixes ere introduced by a refactoring [1]. Kim et al. show how to classify file changes as buggy or clean using change information features and source code terms [12].

Furthermore, other articles have used the SZZ algorithm to measure quality assurance in source code. Matsumoto et al. study the effect of developer features on software reliability resulting that the modules touch by more developer contained more faults [14]. Also bug prediction models, that help allocate quality assurance efforts, used the SZZ in their approach such as Kamei et al. who apply it to validate effort-aware the two common bug-prediction models;

(1) Process metrics LOC, (2) package level predictions and file level predictions [10]. Eyolfson use it to study if time of the day and developer experience affect the probability of a commit to introduce a bug [3]. Izquierdo et al. use the SZZ algorithm to see if developers are fixing their own bugs [9].

Other approach to identify the origins of a bug is describes by Sinha et al. [18]. Their technique is not a text-based technique, as the SZZ algorithm, the authors analyzed the effects of bug-fix changes on program dependencies. Taking into account the semantic of the source code they achieved more accuracy identifying the origins of a bug. These two approaches have the similar ideas: (1) find the differences between the bug-fix version of and the previous version of the file to recognize those changes done by bug-fix commit, (2) look back in the code revision history until identify which version touched the lines affected in the bug-fix for the last time.

3 Methodology

Until now, we have carried out an empirical study where 459 issues reports were analyzed. These issues reports are called tickets in OpenStack, and were taken randomly from four of the more actives repositories of OpenStack project, `Nova, Cinder, Neutron and Horizon`. The OpenStack project was particularly of interest because of its highest scope and heterogeneous nature with hundreds of developers (more than 5,000 developers) contributing to provide an infrastructure for the worlds largest brands such as Ericsson, Dell, Nokia or EBAY. OpenStack has about 184,000 tickets, of which more than 144,000 have been closed by more than 6,000 developers in all its history, and it has more than 233,000 commits with more than 2 million lines of code[1]. All its history is saved and available in a version control system[2], as well as its issue tracking system (Launchpad[3]) and the source code review system (Gerrit[4]). These statistics have been extracted from the publicly available OpenStack database[5].

The study consists of two stages in which at the end we obtain a classification. In the first, three researchers with programming knowledge have worked in parallel analyzing and classifying tickets using a double blind review process. In the second, only one researcher analyzed the cause of the bug.

[1] `http://activity.openstack.org/dash/browser/`
[2] `https://wiki.openstack.org/wiki/Getting_The_Code`
[3] `https://launchpad.net/openstack`
[4] `https://review.openstack.org/`
[5] `http://activity.openstack.org/dash/browser/data/db/`

3.1 First Stage

At this stage, we have to identify what issues found in the Launchpad of each repository are bug reports. This is not a trivial task due to we have to extract information from different places such as Launchpad, Gerrit or Git; and it has to be done manually to ensure a useful classification. But, as the process is repetitive, we developed a web-based tool[6] that helps in this classification process. The tool offers all relevant information required to decide if an issue corresponds to a bug report or not. The tool extracts automatically the information 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 analysis 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.

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 or because of the complexity of the issue .

3.2 Second Stage

In this second stage, we only focused on analyzing the previous commit for those tickets classified in the *Bug Report* group. Therefore we had to locate the buggy line or lines, find out the reason of the software failure, and gathering additional information on the context of the project.

To locate the bug seeding moment, we analyzed the lines involved in the bug fix in two different versions of the file or files where they were; first in the bug fix commit and second, in the *parent* commit of the bug fix commit, in the figure 1 *V4*. We refer to *parent* commit as the commit that modified any line of code in the file before the fix-bug commit was done. And this commit usually is different from the *previous* commit in which the modified lines were the same than in the fix-bug commit. The lines modified in the parent commit do not have to be the ones that have been modified in the bug-fix commit, could be independent commits.

[6] `bugtracking.libresoft.es`

This process ensures us that we are looking the correct change, the change in where the bug was inserted. Because sometimes although the bug seeding commit added many lines, if you look the code before this commit you can check that some of the lines added was there, meaning that came from older commits, 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.

We had to discard some *noise* present in our results, some changes in the previous commit could not have caused the bug and we deleted this previous commit such as blank lines, changes in the format, copied lines, changes in source code comments or updates in the version number of a file/software. Finally, we got a list with all the previous commit and we have to look in all those of them whether the bug was inserted in the previous commit and caused the failure.

4 Preliminary 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.

4.1 Fist Stage

Three researchers including me, were involved in the first stage, classifying a total of 459 tickets using the tool and carried out a double bind, meaning that each ticket were analyzed by two researchers. The Table 1 shows the classification obtained by each developer after analyzed the tickets; not all the developers analyzed the same number of tickets. 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.

After this, we can answer the first research question because at this moment we have all the data necessary and all the knowledge to can distinguish bug reports from others reports. We have obtained that in at least 72% of the tickets analyzed the bug-fixes were real bug reports.

4.2 Second Stage

Only 189 of the tickets classified as bug reports by both researchers were considered in this stage; At this point we have to analyze whether the previous

	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 1. 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.

commit caused the bug or not, in some cases we were unable to decide if this previous commit caused or not the bug.

We have in mind that that a bug-fix could lead changes in many different lines, which could be inserted in different moments by different previous commits. But in fact, not all them may have caused the bug. Hence, the cause could be found in a single previous commit, in many or even in none. But, we need remove some false positives such as blank spaces/lines, changes in the comments or copied lines from previous commits.

At the end, we were be able to identified a total of 348 previous commits, that according with the current assumption caused the 189 bug reports. After discarding the false positives we had to analyze 308 previous commits; Resulting that 152 (49%) of them caused the bug and 114(37%) didn't cause the bug, whereas we have been unable to decide in 42 (14%) cases.

To understand the complexity of identifying the cause of a bug, we focused on measure how many previous commits had each bug reports analyzed. The results are showed in Table 2; obtaining that 131 only had a previous commit, whereas 58 had more than one previous commit.

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

Finally, answering the second research question we can demonstrate that 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

5 Possible applications and further research

In the results we have observed that the current assumption, based on the SZZ algorithm, does not hold in all cases. Around half of the previous commits analyzed have been identified as the cause for inserting the line with the bug. At this moment we know that exist two types of bug seedings, the ones where the previous commit inserted the bug and those in which the bug were inserted by other actions such as changes in the API. Furthermore, one of the challenges in my PhD is going in depth to understand how the bug is inserted in the source code, and what we can do to prevent the bug. Focusing in find an approach that could be automatize and alert to developers in code review process or in automatic inspection process, in these cases where a clean line at some point in the evolution of the code, could be buggy at other point of the source code.

The next researches attempt to answer the RQ3, classifying all the bugs which have not been prevented using either machine learning classifiers or support vector machine [17, 16] based on SZZ algorithm to find a pattern. We could provide a classification of all these bugs which were not prevented, as other previous studies have done [12, 10, 14, 8, 7], according to the following variables to avoid fault prone bug fixes:

- The date of the changes that inserted a bug according to SZZ algorithm.
- The type of changes done in the seeding of the bug such as changes in a function, changes in an If/Else condition, changes in a name variable and so on.
- The number of lines modified in the bug-fix commit.
- The experience in the project of the developer that fix the bug.

This classification could contribute to understand if one of the new variables studied is more prone to appear in these kind of bugs. Meaning that in all these cases where the previous commit didn't injected the bug; is there a pattern repeating?. Currently and after the study presented in this paper, our hypothesis supports the idea of the changes in the API are one of the most prone bugs which are not been inserted by the previous commit, and we could try to prevent these type of bugs in a further study. But the difficulty of this next study reside in how we can prevent the bugs inserted in a change whereas we cannot recognize if this change will be a bug in the future. To answer the RQ4, we must to study which classifier could be the best to prevent this futures bugs.

Also, we extend this study to other projects and other languages such as Java or Java Script due to the compiler interpreter is different in these languages. That way we can ascertain that what we have found occurs regardless of the project analyzed, because the OpenStack may be a special project, where the code is evolving continuously, also the language used is python. In fact, we can access a huge database of other projects such as Eclipse or Mozilla that will allow us analyze this assumption, and where we will use statistical analysis to compare the results obtained in the different projects.

In addition, we want to continue developing the tool, and we will try to automatizes second stage. In addition, could be a good idea develop an automatic classifier based on keywords extracted from the issues tracking systems and code review systems that can distinguish bug reports from other issues. An article about the tool has been accepted in the OSS 2016 conference.

References

1. G. Bavota, B. De Carluccio, A. De Lucia, M. D. Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 104–113. IEEE, 2012.
2. 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.
3. 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.
4. 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.
5. M. Fischer, M. Pinzger, and H. Gall. *Analyzing and relating bug report data for feature tracking.* IEEE, 2003.
6. 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.
7. K. Fujiwara, K. Fushida, N. Yoshida, and H. Iida. Assessing refactoring instances and the maintainability benefits of them from version archives. In *Product-Focused Software Process Improvement*, pages 313–323. Springer, 2013.
8. H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 200–210. IEEE Press, 2012.
9. 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.
10. 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.
11. Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 196–204. IEEE, 2007.
12. 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.

13. 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.
14. S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 18. ACM, 2010.
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. S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *Software Engineering, IEEE Transactions on*, 39(4):552–569, 2013.
17. S. Shivaji, J. E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve bug prediction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 600–604. IEEE Computer Society, 2009.
18. 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.
19. J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.
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.