# What if a Bug has a Different Origin? Making Sense of Bugs Without an Explicit Bug Introducing Commit

Gema Rodríguez-Pérez
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, Spain
gema.rodriguez@urjc.es

Andy Zaidman
Delft University of Technology
Delft, The Netherlands
a.e.zaidman@tudelft.nl

Alexander Serebrenik
Eindhoven University of Technology
Eindhoven, The Netherlands
a.serebrenik@tue.nl

Gregorio Robles
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, Spain
grex@gsyc.urjc.es

Jesús M. González-Barahona
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, Spain
jgb@gsyc.urjc.es

## ABSTRACT

**Background:** Many studies in the software research literature on bug fixing are built upon the assumption that "a given bug was introduced by the lines of code that were modified to fix it", or variations of it. Although this assumption seems very reasonable at first glance, there is little empirical evidence supporting it. A careful examination surfaces that there are other possible sources for the introduction of bugs such as modifications to those lines that happened before the last change an changes external to the piece of code being fixed. **Goal:** We aim at understanding the complex phenomenon of bug introduction and bug fix. **Method:** We design a preliminary approach distinguishing between bug introducing commits (BIC) and first failing moments (FFM). We apply this approach to Nova and ElasticSearch, two large and well-known open source software projects. **Results:** In our initial results we obtain that at least 24% bug fixes in Nova and 10% in ElasticSearch have not been caused by a BIC but by co-evolution, compatibility issues or bugs in external API. Merely 26–29% of BICs can be found using the algorithm based on the assumption that "a given bug was introduced by the lines of code that were modified to fix it". **Conclusions:** The approach allows also for a better framing of the comparison of automatic methods to find bug inducting changes. Our results indicate that more attention should be paid to whether a bug has been introduced and, when it was introduced.

## CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Empirical software validation**;

## KEYWORDS

Bug-introducing change, SZZ algorithm, Empirical Study


*ESEM'18, October 2018, Oulu, Finland*

ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/https://doi.org/10.1145/3239235.3267436

**ACM Reference Format:**
Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona. 2018. What if a Bug has a Different Origin? Making Sense of Bugs Without an Explicit Bug Introducing Commit. In *Proceedings of International Symposium on Empirical Software Engineering and Measurement (ESEM'18).* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/https://doi.org/10.1145/3239235.3267436

## 1 INTRODUCTION

Fixing a bug[1] in a software component usually consists of determining why it is behaving erroneously, and then correcting the part of the component that causes that erroneous behavior. The developer fixing the bug produces a change to the source code, which can be clearly identified as the bug-fixing change (BFC). Identifying which change(s) introduced the bug has proven to be a difficult task.

However, identifying changes that introduce bugs allows, for example, to determine why and how the bug was introduced, which may help, among others, to identify other potential bug introducing changes; to find patterns of bug introduction that could lead to techniques to avoid them; to identify who was responsible for introducing the bug, which could lead to interesting self-learning and peer-assessment processes; to learn about how long the bug was present in the code, which allows in quality assessment. For these and other reasons, finding the changes that introduced bugs is an area of active research during the last decade.

To identify such a change, it is possible to navigate back in the history of the software component, to find out when the malfunction was happening for the first time. This way, we introduce the concept of "first failing moment" to extend the notion of "introducing", we will refer to this moment as *FFM*. Using it, we theorize to determine if a given change was the first one to show a malfunction. From there, we show cases when the *FFM* is the bug-introducing change (*BIC*) and when it is not, in the sense that this change did not cause the malfunction.

In this *emerging results paper* we develop a **preliminary approach** to bug introduction and **initial methodology** for determining whether a given change to the source code is introducing a given fixed bug in a causal sense. Subsequently, we also create a preliminary model of root causes for those bugs for which we

[1]Through this paper we will use term "bug" to refer to the cause of an incorrect result of a software component. In the literature, these causes are also referred to as "defects" or "errors".

have not been able to establish the BIC. Using this approach we can better understand the limitations of both manual and automatic BIC-identification and answer following

> **RQ:** What is the share of bug reports for which we cannot (manually or automatically) find a correct BIC?

The remainder of this paper is structured as follows. We discuss related work and its shortcomings in Section 2, and introduce our approach of bug introduction in Section 3. We apply the approach to two large open-source systems described in Section 4 and report on the first results obtained in Section 5. Finally, we draw conclusions and point out the potential future work in Section 6.

## 2 RELATED WORK

The best known algorithm for automatic identification of code changes introducing bugs was proposed by Śliwerski *et al.* [13]. This algorithm, known as SZZ, has been used in 187 follow-up studies and cited in more than 590 studies as of February 2018 [12]. SZZ is based on text differences to discover modified, added and deleted lines between the BFC and its previous version. The original SZZ algorithm used the CVS `annotate` command[2] to identify the last commit that *touched* these lines. Since the inception of SZZ two main improvements have been proposed: annotation graphs instead of CVS `annotate` [6]; and refined classification of the types of fix-inducing changes [16].

Researchers have largely used SZZ to predict [7, 17], classify [5, 10] and find bugs [4, 15]. Both the original SZZ and the improved versions rely on characteristics of the code at the time of bug fixing because this information can be easily retrieved. However, this practice does not ensure correct identification of when and where the bug was inserted. Recent studies have demonstrated several limitations of SZZ. Rodriguez-Perez *et al.* studied the use and impact of this algorithm and quantified its limitations [12], and Da Costa *et al.* have made an important effort evaluating the results of five alternative SZZ implementations using a proposed a framework [2]. Prechelt and Pepper offered a good overview of the limitations of these methods to identify BICs when are adopted by practitioners [11]. German *et al.* [3] highlighted that changes in software may have impact across the whole system and lead to the manifestation of bugs in unchanged parts. Chen *et al.* studied the characteristics of dormant bugs in the source code [1].

As opposed to the previous studies that have highlighted the limitations of SZZ or tried to improve this algorithm [2, 8] this paper proposes a more reliable alternative process of deciding whether given a BFC a BIC exists, and how to identify the BIC when it exists.

## 3 TOWARDS AN APPROACH TO IDENTIFY BUG-INTRODUCING CHANGES

### 3.1 Description of the preliminary approach

In order to discover the BIC with maximum accuracy, it is recommended to manually backtrack each line of source code that is changed in a BFC until the *moment* when the bug was inserted the first time. If based on the information from the version control system this *moment* coincides with a commit, i.e., a change that can be witnessed in the version control system to either source code or configuration files, the change can be regarded as the BIC. However, in some situations, the failure is not directly caused by a change visible in the version control system, but rather, the failure is due to changes in the context or the environment. The moment when the failure manifests itself is the FFM. In some, but not all cases the BIC coincides with the FFM.

Theoretically the process of identifying the moment when the bug was inserted for the first time, and identifying it as BIC or FFM can be fully automated if the Test Signaling a Bug *(TSB)* is present. TSB is a hypothetical test that could be run on any snapshot[3] of the code and replicate the circumstances under which the bug manifests itself. TSB returns *True* when the test is passed (meaning that the snapshot does not contain the bug) or *False* when the test is not passed (meaning that the snapshot contains the bug). Assuming TSB is available, there is an easy way to find the BIC and FFM by looking manually for the first snapshot in the linear version precedence[4] when TSB fails. This snapshot represents the change that is the perfect candidate to be the BIC or the FFM.

This approach focuses on the cases when given a BFC a BIC can be found or one can be sure that a BIC does not exist. To simplify, we assume that there is only the master branch in the repository of a project. Considering that the TSB can be run indefinitely across the history of the source code, the proposed approach is able to find the BIC and the FFC of a BFC by analyzing the changes that fixed the bug. The TSB will be applied to all the ancestor commits[5] of the BFC, looking for the snapshot that fails; when found, the approach will consider it as a candidate for the BIC or FFM.

### 3.2 Manual Analysis

The input of this stage is a set of bug reports that describe a *real* bug. In addition, bug reports have to be closed and a BFC merged in the code source of the project to be able to apply our methodology.

*3.2.1 Finding the lines that fixed the bug.* We have to find the source code that fixed the commit. We start by identifying the BFC. In many open source projects developers provide a link to the BFC in the bug report's discussion thread, e.g., GitHub encourages this behavior by automatically closing bugs if the commit message contains the bug number preceded by #. Next we determine the lines that the BFC changed. Also typically connected to the bug report, is a link to the code review done for the changes. This information can be essential to determine the origin of the bug as the experts of the project, developers and reviewers, sometimes discuss about the original cause of the bug. By applying *git diff* we can identify what lines have been added, modified or deleted between the version after the BFC and the previous one. Finally, we filter out the added, modified or deleted lines that do not contain source code (e.g., comments or blank lines) are not considered.

*3.2.2 Determining, for each of those lines, the immediate previous commit.* Each individual line touched by the BFC has at most one previous commit, i.e., the most recent commit preceding BFC that has touched the line. However, there could potentially be as many

[2] Other control version systems provide similar functionality, e.g., git offers `blame`.

[3] It represents the entire state of the project at some point in the history

[4] A linear version precedence is constituted by all the ancestor commits of a BFC in a repository

[5] The ancestor commits are all the commits previous to a commit which has been committed into the control version system of a project.

previous commits as modified lines in the BFC. Thus, we talk about the set of all previous commits of the lines of a BFC, the Previous Commit set (PCset).

*3.2.3 Analyze each of these previous commits to determine if it is the BIC.* This analysis uses information available in the description of the ticket and from the logs of the BFC and of all commits in the PCset. Following Śliwerski *et al.* [13], commits after the bug was opened are to be removed. As a result, when we consider the *previous commit* (pc) in order to determine the BIC, one of the following cases holds:

(1) $pc = BIC$, i.e., the previous commit of the line was the cause of the bug. Thus, this commit was the BIC;
(2) $pc \neq BIC$, but the BIC exists: the previous commit was not the BIC, but we are able to identify it in the chain of ancestor commits of the BFC;
(3) $pc \neq BIC$, and the BIC does not exist: the last change was not the BIC and it does not exist in the chain of ancestor commits of the BFC.

## 4 EXPERIMENT

To perform a first evaluation of the preliminary approach of Section 3 we apply it to two large and well-known open source systems, Nova and ElasticSearch. Both systems have been considered in empirical software engineering studies [9, 14].

Nova belongs to OpenStack project. OpenStack is a cloud computing platform mainly written in Python, with a huge development community thereby making it an interesting project to study. It has more than 7,900 contributors and significant industrial support from companies such as Red Hat, IBM, HP. Nova has more than 29,000 commits[6]. All its history is saved and available in a version control system, an issue tracking system[7] and a source code review system[8].

ElasticSearch is a distributed open source search and analytics engine written in Java. It is continuously evolving since its first release in 2010 and currently counts with more than 3,900 commits. This project was chosen because it has a very strict policy of labeling issues, because it has a similar number of commits to OpenStack and because the programming language is different from Nova. The code and issue list of ElasticSearch is hosted in GitHub[9]. There are 1,000 distinct authors and more than 4,500 closed bug reports.

To identify when the bugs appeared and when they were inserted in Nova and ElasticSearch, we manually analyzed the BFC looking for the moment of introducing the bug. In case it does not exist, we investigate and discuss the reasons.

## 5 INITIAL RESULTS

In order to find the BIC, we manually analyzed 116 randomly selected bug reports, 58 from Nova and 58 from ElasticSearch. Each bug report and the associated BFC have been classified in one of the three categories: the BIC exists (corresponding to cases 1 and 2 in Section 3.2.3), the BIC does not exist (corresponding to case 3 in Section 3.2.3), and we could neither find a BIC nor a valid reason for it not to exist (also corresponding to case 3 in Section 3.2.3). The manual analysis has been carried out by the first author that has consulted other authors in case of doubt.

**Table 1: Number of BFCs (Bug-Fixing Commits) caused by a BIC (Bug-Introducing commit) and NoBIC from 116 bug reports from Nova and ElasticSearch.**

| | BIC in BFC | NoBIC in BFC | Unknown |
|---|---|---|---|
| Nova | 38 (65%) | 14 (24%) | 6 (10%) |
| ES | 45 (77%) | 6 (10%) | 7 (12%) |

**Table 2: Reasons why a BFC (Bug-Fixing Commit) is not induced by a BIC (Bug-Introducing commit)**

| | Nova | ElasticSearch |
|---|---|---|
| Co-evolution Internal | 7 (50%) | 2 (33%) |
| Co-evolution External | 2 (14%) | 2 (33%) |
| Compatibility | 1 (7%) | 0 (0%) |
| Bug in External API | 4 (29%) | 2 (33%) |

Table 1 indicates that the BFC is caused by a BIC in 77% for ElasticSearch and 65% for Nova.

When the BFC does not lead to the identification of a BIC, we look for the bug's root cause to find a reason why BIC would not exist. Table 2 shows the main reasons for a bug not having a BIC:

- *Co-evolution Internal*: Changes in the source code that are related to satisfying the new requirements for the project. Since internal resources have been modified (e.g., directory structure and permissions) or requirements have changed, previous assumptions might be invalidated. The bugs caused by the internal co-evolution can be explained by observing that some parts of the code have been updated to reflect the new circumstances, but not all. As such, the changes made in the bug fixing commit cannot be traced back to the BIC.
- *Co-evolution External*: Bugs that are caused by an external change. External resources have been modified without a notification and the project starts to fail.
- *Compatibility*: Bugs are caused by an incompatibility between software and hardware or an incompatibility with an operating system.
- *Bug in External API*: A change in the API of a third-party code caused a bug in the source code of the project.

Finally, if we neither found BIC nor a reason for it not to be present we classify the ticket as *Unknown*. Reasons for us not being able to find BIC can be related to lack of information or to presence of consecutive changes in the affected lines that make it impossible to track it back to find the BIC.

Table 3 shows the percentage of BICs that can be found using the SZZ-1 algorithm[10]. In order to find the BIC, we manually analyzed the previous commits in all the PCset of the 116 random bug reports. Since the PCset of a bug report can be greater than 1, the total

[6] http://stackalytics.com
[7] https://launchpad.net/openstack
[8] https://review.openstack.org/
[9] https://github.com/elastic/elasticsearch/

[10] We denote SZZ-1 as the enhancement of SZZ by Kim *et al.* [6]. We assume that when a BFC presents a PCset greater than 1, the SZZ-1 flags all of them as BICs. Other implementations are also possible, however, they will be studied in the future work

**Table 3: Number of bug-fixing commits with BIC and no BIC from 116 bug reports**

| | $pc = BIC$ | $pc \neq BIC$ | Unknown |
|---|---|---|---|
| Nova | 36 (26%) | 83 (60%) | 18 (13%) |
| ES | 35 (29%) | 64 (53%) | 22 (18%) |

number of previous commits analyzed are 120 in Nova and 137 in ElasticSearch. We observe that from the previous commits analyzed in Nova and ElasticSearch, only a 26%-29% of the results can be identified as BIC by the SZZ-1 algorithm. The mean of previous commits per BFC in Nova is 2.3, while in ElasticSearch this number is lower, 2.06. For this reason, the percentage of $pc \neq BIC$ in both projects is relatively high.

Some of the reasons why a pc analyzed was not the BIC are: (1) The BFC removed obsolete code, (2) The BFC optimized some lines of the code, (3) The line modified in the BFC already contained the bug as it was inserted in a previous commit of this line, (4) The BFC modified lines that were not related with the bug, and (5) The BFC modified a line that was not buggy at the time of committing it.

**The percentage of a BIC causes a BFC is around 65%-77% in both projects, whereas 10%-24% of the BFC do not present a BIC. The number of BICs that can be found using the SZZ-1 algorithm is 26%-29%.**

## 6 CONCLUSIONS AND FUTURE WORK

After analyzing 116 BFC and a set of 257 previous commits, we have realized that determining where and when a bug was introduced is not a trivial task. In fact, even if the location and time stamp of a BIC are known, we need to (1) understand the bug and how to fix it in the case we have indeed found the actual BIC, or (2) determine that what we assume to be the BIC, is in fact not the BIC, but simply the FFM because the bug was not present at this moment.

Our data clearly shows that identification of BIC is impossible using solely techniques that rely on backtracking the source code lines that have been modified to fix the bug. Indeed, BIC does not exist in 10% in ElasticSearch and 24% in Nova, and even when BIC exists, these techniques make assumptions which can cause the wrong identification of BICs: our first results indicate that a 53%-60% of the previous commit analyzed are not the BIC. This highlights the inherent limitations of any SZZ-based technique and seed some lights to investigate other techniques different from tracking back the lines fixed in the BFC.

Bugs with no BICs might be caused by external changes to the project or the modification of internal resources. In such circumstances to fix the bug, developers might be required to change lines that were correct at the time of writing, but are no longer correct. These lines, hence, manifested the bug, but did not introduce it.

Our first results should be seen as a call for a new technique for identification of BICs that can deal with complexity of bug kinds. As such, this paper proposes a preliminary approach which relies on the implementation of the TSB to locate the BIC and the FFM. This approach needs to be further studied and formalized to better understand the complex phenomenon of bug introduction.

For future work, our primary action is to formalize and further validate the proposed approach. The full automation of the methodology used in this paper is also interesting from a practical point of view. That would provide software projects with a valuable tool for understanding how they are introducing bugs, and therefore design measures for mitigation. It would also help to improve the performance of the current state-of-the-art tools and techniques. Moreover, software developers can benefit from identifying where the bug was inserted, improving their processes. Finally, the study whether the percentage of BICs depends on the programming language used in the project would be also interesting to look at.

## REFERENCES


[1] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. 2014. An empirical study of dormant bugs. In *MSR*. ACM, 82–91.

[2] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uira Kulesza, Roberta Coelho, and Ahmed Hassan. 2016. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Trans. on Softw. Engineering* (2016).

[3] Daniel M. German, Ahmed E Hassan, and Gregorio Robles. 2009. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology* 51, 10 (2009), 1394–1408.

[4] Sunghun Kim, Kai Pan, and EE Whitehead Jr. 2006. Memories of bug fixes. In *FSE*. ACM, 35–45.

[5] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Trans. on Softw.. Engineering* 34, 2 (2008), 181–196.

[6] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead Jr. 2006. Automatic identification of bug-introducing changes. In *ASE*. IEEE, 81–90.

[7] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *ICSE*. IEEE, 489–498.

[8] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The Impact of Refactoring Changes on the SZZ Algorithm: An Empirical Study. In *SANER*. IEEE, 380–390.

[9] Matteo Orrú, Ewan Tempero, Michele Marchesi, Roberto Tonelli, and Giuseppe Destefanis. 2015. A Curated Benchmark Collection of Python Systems for Empirical Studies on Software Engineering. In *PROMISE*. ACM, Article 2, 4 pages.

[10] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. 2006. Bug classification using program slicing metrics. In *SCAM*. IEEE, 31–42.

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

[12] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. 2018. Reproducibility and Credibility in Empirical Software Engineering: A Case Study based on a Systematic Literature Review of the use of the SZZ algorithm. *Information and Software Technology* (2018).

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

[14] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G. J. van den Brand. 2014. Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub. In *ICSME*. IEEE, 401–405.

[15] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *ASE*. IEEE, 262–273.

[16] Chadd Williams and Jaime Spacco. 2008. SZZ revisited: verifying when changes induce fixes. In *Workshop on Defects in large software systems*. ACM, 32–36.

[17] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for Eclipse. In *PROMISE*. IEEE, 9.