How bugs are born
a model to identify how bugs are introduced in software components

Rodríguez-Pérez, Gema; Robles, Gregorio; Serebrenik, Alexander; Zaidman, Andy; Germán, Daniel M.; Gonzalez-Barahona, Jesus M.

DOI
10.1007/s10664-019-09781-y

Publication date
2020

Document Version
Final published version

Published in
Empirical Software Engineering

Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09781-y

How bugs are born: a model to identify how bugs are introduced in software components

Gema Rodríguez-Pérez[1] · Gregorio Robles[2] · Alexander Serebrenik[3] · Andy Zaidman[4] · Daniel M. Germán[5] · Jesus M. Gonzalez-Barahona[2]

Published online: 04 February 2020

Abstract
When identifying the origin of software bugs, many studies assume that "a bug was introduced by the lines of code that were modified to fix it". However, this assumption does not always hold and at least in some cases, these modified lines are not responsible for introducing the bug. For example, when the bug was caused by a change in an external API. The lack of empirical evidence makes it impossible to assess how important these cases are and therefore, to which extent the assumption is valid. To advance in this direction, and better understand how bugs "are born", we propose a model for defining criteria to identify the first snapshot of an evolving software system that exhibits a bug. This model, based on the *perfect test* idea, decides whether a bug is observed after a change to the software. Furthermore, we studied the model's criteria by carefully analyzing how 116 bugs were introduced in two different open source software projects. The manual analysis helped classify the root cause of those bugs and created manually curated datasets with bug-introducing changes and with bugs that were not introduced by any change in the source code. Finally, we used these datasets to evaluate the performance of four existing SZZ-based algorithms for detecting bug-introducing changes. We found that SZZ-based algorithms are not very accurate, especially when multiple commits are found; the F-Score varies from 0.44 to 0.77, while the percentage of true positives does not exceed 63%. Our results show empirical evidence that the prevalent assumption, "a bug was introduced by the lines of code that were modified to fix it", is just one case of how bugs are introduced in a software system. Finding what introduced a bug is not trivial: bugs can be introduced by the developers and be in the code, or be created irrespective of the code. Thus, further research towards a better understanding of the origin of bugs in software projects could help to improve design integration tests and to design other procedures to make software development more robust.

Keywords Bug origins · Bug-introducing changes · First-failing change · SZZ algorithm · Extrinsic bugs · Intrinsic bugs

Communicated by: Per Runeson

✉ Gregorio Robles
grex@gsyc.urjc.es

Extended author information available on the last page of the article.

1 Introduction

During the life of a software product developers often fix bugs[1] (Pan et al. 2009; Murphy-Hill et al. 2015). Research has shown that developers spend half of their time fixing bugs; while they devote only about 36% to adding features (the rest goes to making code more maintainable) (LaToza et al. 2006). Fixing a bug consists of determining why software is behaving erroneously, and subsequently correcting the part of the component that causes that erroneous behavior (Zeller 2009; Beller et al. 2018; Beller et al. 2015; Ebert et al. 2015). A developer fixing a bug produces a change to the source code, which can be identified unambiguously as the bug-fixing change (***BFC***). However, identifying what change(s) introduced the bug has proven to be a more difficult task (da Costa et al. 2017; Rodríguez-Pérez et al. 2018a).

Nonetheless, identifying the changes that introduced bugs would enable to (1) discover bug introduction patterns which could be used to develop techniques to avoid changes introducing bugs (Hassan 2009; Hassan and Holt 2005; Kim et al. 2007); (2) identify who was responsible for introducing the bug for the sake of self-learning and peer-assessment (Izquierdo-Cortazar et al. 2011; da Costa et al. 2014; Ell 2013); or (3) understand how long the bug has been present in the code (e.g., to infer how many released versions have been affected or how effective the project testing/verification strategy is (Rodriguez-Perez et al. 2017; Chen et al. 2014; Weiss et al. 2007)). For these, among other reasons, identifying what changes introduced bugs has been a very active area of research over the last decade (Abreu and Premraj 2009; Aranda and Venolia 2009; da Costa et al. 2017).

The vast majority of this research is based on the assumption that *a bug was introduced by the lines of code that were modified to fix it* (Śliwerski et al. 2005; Kim et al. 2006; Williams and Spacco 2008). Although the literature frequently uses this assumption, there is not enough empirical evidence supporting it. Indeed, recent studies have demonstrated that well-known algorithms based on this assumption (such as the approach proposed by Sliwerski, Zimmermann, and Zeller (SZZ) (2005)) tend to incorrectly identify the bug-introducing changes (***BIC*s**) (da Costa et al. 2017; Rodríguez-Pérez et al. 2018a). For some bugs an explicit change introducing it does not even exist; the system behaves incorrectly due to changes that are external to the system (German et al. 2009; Rodríguez-Pérez et al. 2018b).

In this work we focus on analyzing how bugs were introduced in a software component, therefore we evaluate whether the aforementioned assumption holds.

For a major part, this work has been possible because in modern software development the history of a software product is typically recorded in a source code management (***SCM***) system, which enables researchers to retrieve and trace all changes to its source code, and understand the reasons why a change fixed a bug.

We selected two open source projects, Nova and ElasticSearch, as exploratory case studies to understand and locate, whenever possible, what change(s) introduced bugs and their characteristics. We analyze those cases in which a *BFC* in the SCM of Nova and ElasticSearch can be associated with a bug. To accomplish this task, we identify bugs in the system

[1] Throughout this paper, we use the term "bug", which we define in detail in Section 5. Although bugs could be considered as "defects/faults" or "failures", according to Institute of Electrical and Electronics Engineers and IEEE Computer Society. Software Engineering Standards Committee (2009) and ISO/IEC (2001), we use "bug" as it is widely used in the literature (Tan et al. 2014; Śliwerski et al. 2005; Chen et al. 2014) and it is the term developers use normally. We also describe shades between all these terms in Section 2.

using the issue tracker system (***ITS***) (bugs that were fixed directly in the source code without an entry in the bug tracker system (Aranda and Venolia 2009) are outside the scope of this research). The ITS links directly to the change (commit) that fixed the bug (its *BFC*). Using this information, we will navigate back the history of the source code to identify the origin for each of the bugs in both case studies.

1.1 Goal: A Model of How Bugs Were Introduced

Based on this analysis, we propose a model of how bugs were introduced, from which the assumption that *a bug was introduced by the lines of code that were modified to fix it* can be derived as a specific case. The model classifies bugs into two categories: (1) **intrinsic bugs**: bugs that were introduced by one or more specific changes to the source code; and (2) **extrinsic bugs**: bugs that were introduced by changes not registered in the SCM (e.g., from an external dependency), or changes in requirements.

The proposed model will be of help in the complex task of identifying the origin of bugs, particularly, the idea of the "perfect test". This idea is fundamental (1) to decide whether a **snapshot**[2] of a software component is affected by a bug; and (2) to identify which version of a software component exhibits the bug for the first time. Furthermore, this model is necessary for two main reasons: (1) its application in real-world cases provides the formalisms (e.g., definitions) to create a manually curated dataset with bug-introducing changes, when they exist; and (2) it can precisely define criteria to decide the first manifestation of a bug in the history of an open source software product.

The current absence of such criteria causes ambiguity of what snapshot should be considered as "exhibiting a bug", which renders any approach to find the *BIC* arguable. For example, software may work properly until the system where it runs on upgrades a library it depends on (an event that might not be recorded in version control). Note that in this scenario the same snapshot does not exhibit the bug before the library upgrade, but exhibits the bug after.

In such a case, the changed lines by the *BFC* were not the cause of the bug (these lines were correct until the upgrade). Our proposed model establishes criteria that allow researchers to determine that the snapshot after the upgrade did not introduce the bug but, it exhibited the bug for the first time.

In the previous example, the snapshot that first exhibited the bug was the one that was run after the library upgrade. However, which snapshot exhibits the bug? The one before the library upgrade, or any version that exhibits the bug after the library upgrade? Currently, there is not a common way to assess that the changes identified as first exhibiting the bug by current approaches (Śliwerski et al. 2005; Kim et al. 2006; Thung et al. 2013) are true/false positives/negatives since they do not have into account this example.

Hence, in this paper, we set out to address the following question:

> *"How can we identify the origin of a defect based on information in source control systems?"*

1.2 Research Questions

In particular, to answer our central question, we first defined specific criteria that help determine whether a change in the source code introduced a bug, and the moment this change was

[2] A snapshot is the state of the system after a commit.

introduced. Then, we studied these criteria in some real-world cases. Thus, we addressed the following research questions (RQs):

- **RQ1**: Is there a criteria to help researchers find a useful classification of changes leading to bugs?
  Motivation: Our designed model provides defined criteria to decide whether a certain bug is present in a snapshot. However, we need to ensure that these criteria can be applied to real-world projects to determine whether a change in the source code introduced a bug. Thus, we used the model to understand and classify the root cause in 116 bugs. This process produced two manually curated datasets that contain a collection of bugs, and information on a) the change to the source code that introduced the bug, or b) the absence of such a change.
- **RQ2**: Do these criteria help in defining precision and recall in four existing SZZ-based algorithms for detecting bug-introducing changes?
  Motivation: The positive answer to RQ1, at least for some cases, helped us create manually curated datasets that may be considered as the "ground truth" for some bugs. We use this "ground truth" datasets to compare four existing SZZ-based algorithms that identify *BICs* and compute their performance (in terms of precision, recall and F-score), and compare them against each other. The analysis of the results helps to find ways to improve them.

1.3 Contributions

This work is a further development of our preliminary work (Rodríguez-Pérez et al. 2018b), which we are extending with the following main results, based on prior literature and empirical findings:

1. A **model** that, given a *BFC*, describes when the corresponding bug was introduced, consisting of (i) a set of explicit assumptions on how bugs were introduced, (ii) specific criteria for deciding whether a bug is present in a snapshot, (iii) a process for determining which change in the source introduced the bug, or the knowledge that it was not introduced by a change, and (iv) a proposed terminology of the components that play a role in the bug introduction process.
2. An **operationalization of the process to determine which change first exhibited the bug** that can be used to (i) classify the bug as intrinsic or extrinsic, (ii) identify the first snapshot that contains the bug.
3. A **unified terminology** with all relevant concepts involved in the origin of bugs. A common terminology is needed because we have found in the literature that scholars use different wording for the same concepts or, even worse, use the same wording for different concepts. This situation hinders the understanding of the bug origin problem and can be solved with a unified terminology.
4. An **empirical study on two open source software systems** (ElasticSearch and Nova) that exemplifies how our model and operationalization can be applied to two real open source projects. The result of this study is a manual curated reference dataset that annotates a set of bug fixing changes with the change that introduced the bug, or with the absence of such a change (in our case we do it for a collection of 116 bug reports).
5. An **evaluation of the performance** of four existing SZZ-based algorithms for the identification of *BICs*. This evaluation provides further insights on how these algorithms could be improved.

The rest of this paper is structured as follows. We first introduce some motivating examples in Section 2 to support the convenience of developing a model to describe how bugs were introduced. Related work is presented in Section 3. Then, we introduce the general framework and the assumptions we consider, in Section 4. Section 5 describes the model, the associated terminology and the process to determine which change first exhibited the bug. Then, Section 6 details the operationalization of these process. Section 7 introduces the case studies and the empirical results. Section 8 discusses potential applications, guidelines and improvements, and reports on threats to validity. Finally, we draw conclusions and point out potential future research in Section 9.

2 Background and Motivation Examples

Software is prone to defects due to its inherent complexity and the developers' difficulties to understand its design (Itkonen et al. 2007). Therefore, defects and how they are introduced in code have been an active area of research (see Basili and Perricone (1984), Mockus and Weiss (2000), and Boehm and Basili (2005) for some seminal work on the matter of understanding and classifying how defects are introduced). According to IEEE Standard 1044 (2009), a defect is "*an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced*". When the defect is present in software, it is considered a "fault" (*manifestation of an error in software*). A defect/fault can be introduced in different phases of a software product life (e.g., planning, coding, deployment) due to many reasons, such as missing or changing requirements, wrong specifications, miscommunication, programming errors, time pressure, poorly documented code, among others (Nakajo and Kume 1991; Jacobs et al. 2007; Nuseibeh and Easterbrook 2000). When the software is executed and the system produces wrong results, defects may lead to failures, described as "*[e]vents in which a system or system component does not perform a required function within specified limits*" (Institute of Electrical and Electronics Engineers and IEEE Computer Society. Software Engineering Standards Committee 2009). Developers, and in many cases researchers too, typically use the term "bug" to refer both to defects/faults (deficiencies) and failures (their manifestation), depending on the context. For example, "fixing a bug" usually means "fixing a failure by correcting the faulty code" while "reporting a bug" means "reporting a failure". A single fault may lead to several failures and, in some cases, a single failure may be caused by several faults. Through this paper, we will use in general the term "bug", trying to specify, when that is relevant and is not obvious from the context, if we refer to failures or faults. We will also assume that when a "bug is fixed" means that "a failure was fixed by correcting at least one fault". In general, we will be interested in the first fault (per order of introduction in the source code), in case there are more that one causing a failure.

However, neither IEEE 1044 (2009) nor ISO/IEC 9126 (2001) provide a way of determining whether some code can be considered buggy (or faulty) *when it was written*. Of course, researchers and developers may know if some code is considered faulty when a certain failure is fixed, but that is not enough to know if it could also be considered faulty when it was written, or at that time it was perfectly correct, according to the context of the system at that moment. The lack of definitions and some previous unconsidered origins[3] for

[3] The current literature does not consider a change that has not been recorded in the SCM of a project (e.g., in requirements, to external APIs, to the environment) as "the origin of a bug".

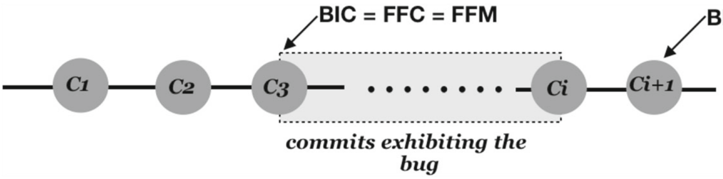


Fig. 1 Intrinsic bug: the bug-introducing change (*BIC*) is recorded in the source code management (*SCM*). The first-failing change (*FFC*) and the first-failing moment (*FFM*) coincide with the *BIC*

bugs (Rodríguez-Pérez et al. 2018b) cause difficulties to correctly identify which change introduced a fault, and even if the fault was introduced by it, or by a later change in the context of the system. Furthermore, with a precise definition of "introducing a fault" (from now on, "introducing a bug"), researchers can identify whether a change that exhibits a given bug is also the change responsible for introducing it (i.e., the bug-introducing change (*BIC*)) or whether this change corresponds to the first time that the system manifested the bug. In other words, the fact that before a given change the system does not exhibit a bug, but after it, the bug appears, is not enough to consider that the change introduced the bug.

We will refer to this later case with the concept of "first-failing change" (***FFC***), in the sense that this change did not introduce the bug, but there was a "first-failing moment" (**FFM**) –not recorded in the SCM– in which the bug manifests itself for the first time. Thus, in this work, when there is an intrinsic bug, the bug-introducing change, the first-failing change and the first-failing moment are the same (see Fig. 1). However, when there is an extrinsic bug, there is no bug-introducing change in the SCM and the first-failing change is the commit in our SCM right after the first-failing moment occurs (see Fig. 2).

Bugs can be classified as (a) intrinsic, when the bug has a bug-introducing change (*BIC*) which coincides with the first-failing change (*FFC*) and the first-failing moment (*FFM*), or (b) extrinsic, when the bug does not have a *BIC* but a *FFC* which differs from the *FFM*.

Extrinsic bugs are caused by changes that are not recorded in the SCM. These bugs are not the result of introducing faulty code, but might be due to incorrect assumptions, changes in requirements, dependencies on the run-time environment, changes to the environment, bugs in external APIs, among others. As far as we know, this kind of bugs has not been studied before from the perspective of their introduction; this work aims to offer more insights into such bugs. In the next examples, we show some extrinsic bugs and motivate the interest in researching them.

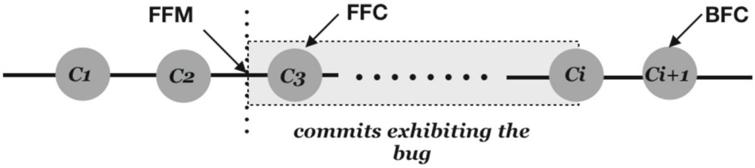
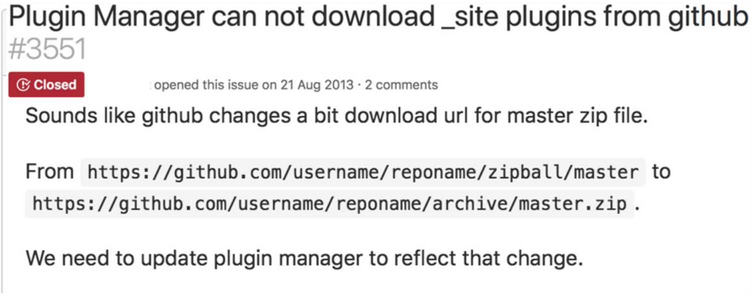


Fig. 2 Extrinsic bug: the first-failing moment (*FFM*) does not coincide with a change in the source code management (*SCM*). There is no bug-introducing change (*BIC*), and the first-failing change (*FFC*) is the first change recorded in the SCM after the first-failing moment *FFM*



Plugin Manager can not download _site plugins from github #3551

opened this issue on 21 Aug 2013 · 2 comments

Sounds like github changes a bit download url for master zip file.

From `https://github.com/username/reponame/zipball/master` to `https://github.com/username/reponame/archive/master.zip`.

We need to update plugin manager to reflect that change.

Fig. 3 ElasticSearch bug report #3551 (Example 1)

Example 1 Figure 3 shows a bug report from the ElasticSearch project.[4] The bug occurred when downloading a site plugin from GitHub. In this case, the dependency of the source code of ElasticSearch on the GitHub API caused the bug. Around seven months after inserting the original lines, the GitHub API changed and the source code in ElasticSearch became buggy because the plugin no longer worked. Figure 4 shows the lines modified to fix the bug. The original version of these lines did not introduce the bug, but they are the lines where the bug manifested itself (after the change in the GitHub API). Thus, there is no change to the source code of ElasticSearch itself that introduced the bug because when those lines were introduced the GitHub API worked as the developer expected. Table 1 summarizes the existence of the bug-introducing change, first-failing change and first-failing moment in this example.

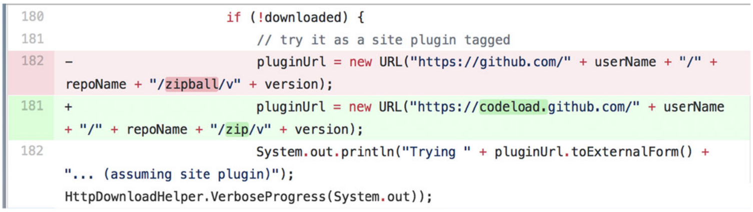
Example 2 Figure 5 offers another bug report from ElasticSearch[5]. This bug pertains to setting permissions in subdirectories; it was caused by the post-installation script setting all data permissions to 644 inside of `/etc/elasticsearch`, and failing to set appropriate permissions (755) to subdirectories. The only line that was modified to fix this bug was line 37 (see Fig. 6). However, as directories did not exist in `/etc/elasticsearch` when the original version of line 37 was introduced, we can conclude that there is no *BIC*. Table 2 summarizes the existence of the bug-introducing change, first-failing change and first-failing moment in this example.

Example 3 Some bugs manifest themselves if the software is used in a different environment than it was intended for. Figure 7 shows a bug report in Nova describing a failure when using Windows Server 2012; Windows Server 2012 introduced support for projecting a virtual NUMA topology into Hyper-V virtual machines. Here, as well, there is no *BIC*, and the manifestation of the bug depends on the environment used. Table 3 summarizes the existence of the bug-introducing change, first-failing change and first-failing moment in this example.

The bug in Example 1 manifested itself due to a change to an external artefact upon which the software depends. The bug in Example 2 manifested itself due to an incorrect assumption (in this case, an omission of a requirement). Example 3 shows a bug

[4] https://github.com/elastic/elasticsearch/issues/3551

[5] https://github.com/elastic/elasticsearch/issues/3820

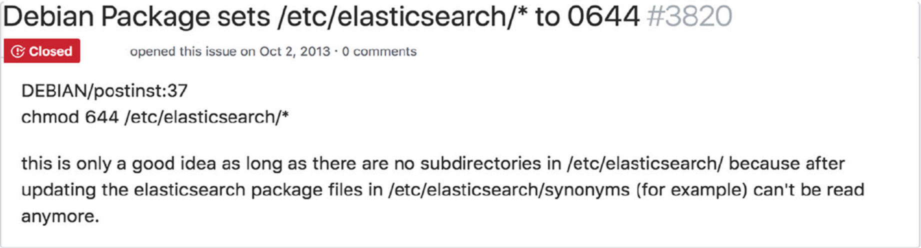


```
                    if (!downloaded) {
                        // try it as a site plugin tagged
-                       pluginUrl = new URL("https://github.com/" + userName + "/" +
repoName + "/zipball/v" + version);
+                       pluginUrl = new URL("https://codeload.github.com/" + userName
+ "/" + repoName + "/zip/v" + version);
                        System.out.println("Trying " + pluginUrl.toExternalForm() +
"... (assuming site plugin)");
HttpDownloadHelper.VerboseProgress(System.out));
```

Fig. 4 Bug Fixing commit of #3551 (Example 1)

Table 1 First-failing moment (FFM), first-failing change (FFC) and bug-introducing change (BIC) in Example 1

	Where	Presence in the SCM
BIC	None (extrinsic bug)	No
FFM	When the GitHub API changed	No
FFC	First commit after the GitHub API changed	Yes



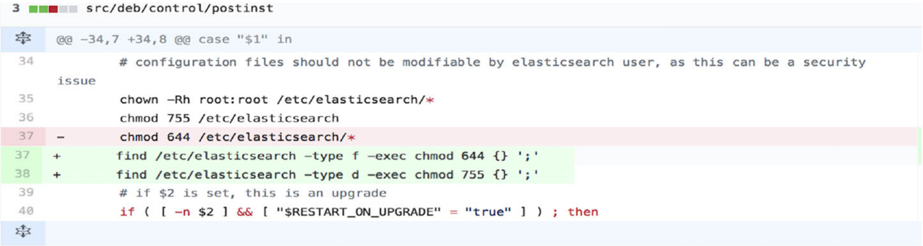
Debian Package sets /etc/elasticsearch/* to 0644 #3820

Closed opened this issue on Oct 2, 2013 · 0 comments

DEBIAN/postinst:37
chmod 644 /etc/elasticsearch/*

this is only a good idea as long as there are no subdirectories in /etc/elasticsearch/ because after updating the elasticsearch package files in /etc/elasticsearch/synonyms (for example) can't be read anymore.

Fig. 5 ElasticSearch bug report #3820 (Example 2)

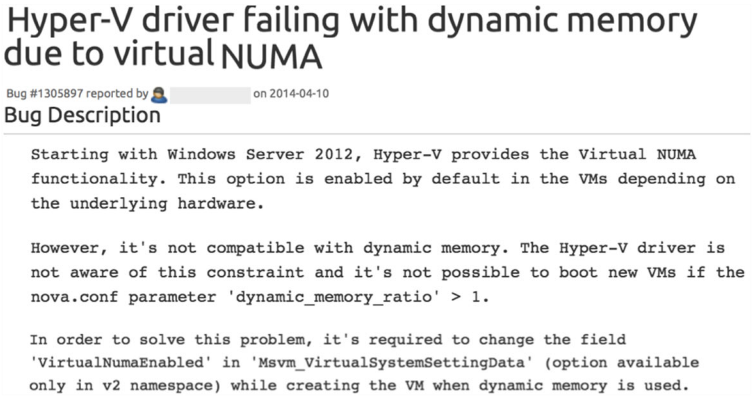


```
3 src/deb/control/postinst
@@ -34,7 +34,8 @@ case "$1" in
        # configuration files should not be modifiable by elasticsearch user, as this can be a security
issue
        chown -Rh root:root /etc/elasticsearch/*
        chmod 755 /etc/elasticsearch
-       chmod 644 /etc/elasticsearch/*
+       find /etc/elasticsearch -type f -exec chmod 644 {} ';'
+       find /etc/elasticsearch -type d -exec chmod 755 {} ';'
        # if $2 is set, this is an upgrade
        if ( [ -n $2 ] && [ "$RESTART_ON_UPGRADE" = "true" ] ) ; then
```

Fig. 6 Bug Fixing commit of #3820 (Example 2)

Table 2 First-failing moment (FFM), first-failing change (FFC) and bug-introducing change (BIC) in Example 2

	Where	Presence in the SCM
BIC	None (extrinsic bug)	No
FFM	When the requirements changed	No
FFC	First commit after requirements changed	Yes



Hyper-V driver failing with dynamic memory due to virtual NUMA

Bug #1305897 reported by [redacted] on 2014-04-10

Bug Description

Starting with Windows Server 2012, Hyper-V provides the Virtual NUMA functionality. This option is enabled by default in the VMs depending on the underlying hardware.

However, it's not compatible with dynamic memory. The Hyper-V driver is not aware of this constraint and it's not possible to boot new VMs if the nova.conf parameter 'dynamic_memory_ratio' > 1.

In order to solve this problem, it's required to change the field 'VirtualNumaEnabled' in 'Msvm_VirtualSystemSettingData' (option available only in v2 namespace) while creating the VM when dynamic memory is used.

Fig. 7 Bug caused by the operating system where the code is being used (Example 3)

caused by a change in the environment, as the bug manifested when the software was used in a platform it did not officially support at the time of writing the code. These cases are examples of extrinsic bugs, in which there is no bug-introducing change causing the bug.

As we can observe, extrinsic bugs are not the result of an explicit change in the SCM. Thus, it is necessary to develop new models to describe their origin.

3 Related Work

Traditionally, in mining software repositories, researchers identify the lines of source code that introduced the bug assuming that the last change that touched the fixed line(s) in a bug-fixing change (*BFC*) introduced the bug (Zeller et al. 2011; Śliwerski et al. 2005; Williams and Spacco 2008). Thus, the introduction of bugs has been studied over the last years from the *BFC* backward by using two different methods: dependency-based and text-based methods.

Dependency-based approaches use changes in the relationship between control and data in the code. Ottenstein and Ottenstein proposed the first program dependence graph to be used in software engineering (Ottenstein and Ottenstein 1984). This approach achieves higher accuracy than text-based approaches (Sinha et al. 2010) in identifying the bug-introducing change (*BIC*), taking into account the semantics of the source code, because it addresses some of the limitations of text-based approaches (Davies et al. 2014). However, dependency-based approaches are not appropriate for identifying the origins of all bugs

Table 3 First-failing moment (FFM), first-failing change (FFC) and bug-introducing change (BIC) in Example 3

	Where	Presence in the SCM
BIC	None (extrinsic bug)	No
FFM	When running Windows Server 2012	No
FFC	The commit that introduced the Virtual NUMA functionality	Yes

because they have some implementation challenges. For instance, these approaches cannot identify the *BIC* when the *BFC*s do not change the method's dependencies.

On the other hand, the text-based approaches are more popular when identifying the *BIC* since they pose less implementation challenges (Davies et al. 2014), thus the related work section focuses on these approaches. Text-based approaches are based on textual differences to discover addition, deletion and modifications lines between the *BFC*s and its previous version, and then backtrack the modification and deletion lines to identify the change that introduced the bug. The approach proposed by Sliwerski, Zimmermann, and Zeller (SZZ) is a popular text-based algorithm (Rodríguez-Pérez et al. 2018a), improving on previous text-based approaches (Čubranic and Murphy 2003; Fischer et al. 2003a; 2003b). As such, it assumes that the last change that touched the fixed line in a *BFC* introduced the bug (Śliwerski et al. 2005) and relies on historical data to identify changes in the source code that introduced bugs. For that, the algorithm links the SCM and the ITS in order to identify the *BFC* and then, it identifies the *BIC*. To that end, it employs the diff functionality to determine the lines that have been changed between the *BFC* and its previous version and the blame functionality to identify the last change(s) to those lines. Finally, it uses a temporary window from the bug report date until the *BFC* date to remove false positives.

Since the inception of SZZ two main improvements have been proposed: Kim et al. used annotation graphs to reduce false positives and gain precision by excluding comments, blank lines, and format changes from the analysis (Kim et al. 2006); and Williams and Spacco improved the line mapping algorithm of SZZ by using weights to map the evolution of a line (Williams and Spacco 2008). Many studies have largely used these SZZ algorithms to predict, classify and find bugs. Kamei et al. proposed a model to identify defect-prone changes instead of defect-prone files or defect-prone packages; this model allows developers to review these risky changes while they are still fresh in their minds, which is known as 'Just-in Time Quality Assurance' (JIT) (Kamei et al. 2013). Kim et al. showed how to classify file changes as buggy or clean using change information features and source code terms (2008). Tantithamthavorn et al. studied how to improve the bug localization performance assuming that a recently fixed file may be fixed in the near future (2013). Nagappan et al. used the SZZ idea of mapping as the base to associate metrics with post-release defects, and built regression models to predict the likelihood of post-release defects for new entities (2006). Zimmermann et al. used the SZZ to predict bugs in large software systems (2007).

Recently, Da Costa et al. have made an important effort proposing a framework for evaluating the results of five SZZ implementations. This framework assesses the data generated by SZZ implementations and flags changes as not likely to be *BIC*s. For that, this framework relies on three criteria: (1) the earliest bug appearance which is related to the number of disagreements that SZZ has with the affected-version reported; (2) the impact that a *BIC* has in future bugs; and (3) the likelihood that the *BIC* given by SZZ is the *real* cause of the bug computed as the difference in days between the first and the last suspicious *BIC*s; if this difference is several years, the commit is removed. Their findings showed that current SZZ implementations still lack mechanisms to correctly identify *real BIC*s (da Costa et al. 2017). In this work, we describe how to use our model to identify *real BIC*s, which is one of the the major problems of SZZ algorithms. While Da Costa et al. base their study on the reliability of SZZ results with computing metrics, our aim is to describe a model that can help to reason about whether an earlier change in the SCM caused the bug.

Furthermore, Campos Neto et al. have studied the impact of refactoring changes on SZZ and have proposed the RA-SZZ implementation (*Refactoring Aware-SZZ*). Refactoring changes are one of the major limitations of SZZ since the algorithm blame them as bug-introducing changes when, in fact, these changes did not introduce the bug because they did not change the system behavior. The authors observed that 6.5% of the lines blamed as *BIC*s by SZZ were refactoring changes and that 19.9% of the lines removed in a *BFC* were related to refactoring changes (2018). In addition, Campos Neto et al. re-evaluated the RA-SZZ implementation in Defects4J dataset and observed that 44% of the lines identified as *BIC*s by RA-SZZ are very like to real *BIC*s. However, there exist refactoring operations (31.17%) and equivalent changes (13.64%) that are misidentified by RA-SZZ (2019). While Campos Neto et al. assumed that the *BIC* should be in the evolutionary history of the lines that have been changed in a *BFC*, our work takes a backward step to understand how bugs were introduced and describe a model that can help with this identification. In our model, the evolution history of the lines that have been changed in a *BFC* can be derived as a specific case of how bugs were introduced.

More recently, Sahal and Tosun proposed a way to link the code additions in a fixing change to a list of candidate *BIC*s (2018). The authors state that their approach works well for linking code additions with previous changes, although it still produces many false positives since this approach assumes that the *BIC* is one of the changes surrounding the new additions in a *BFC*. Our model helps researchers to understand whether an incomplete change caused a bug and then, the *BFC* fixed this bug by adding only new lines of source code. However, our model does not assume that the *BIC*s have to be the changes surrounding the new additions.

In addition, other studies observed serious limitations when using both dependency-based and text-based approaches. These limitations are addressed in the model proposed in this work. Murphy-Hill et al. observed that when developers fix bugs, they have different options as to how to fix them and each decision may lead to a different location where a bug was introduced (2015). Qualitatively, the authors showed the many factors that influence how bugs are fixed, most of them being non-technical. These factors may affect bug prediction and localization because the bug fixing cannot be at the same location as the bug, or because the bug fixing might be covering the symptom and not the cause of the bug. Rodríguez-Pérez et al. performed a systematic literature study on the use of the SZZ algorithm and quantify its limitations (2018a). Prechelt and Pepper offered an overview of the limitations of the text-based approaches when they are used for *Defect-Insertion Circumstance Analysis (DICA)* (2014). The authors observed that *BFC*s may have touched non-buggy lines, and even when they touched those lines, the actual *BIC* may have been made earlier. Also, they stated that bugs and issues are not easy to distinguish in bug trackers, causing low reliability when mapping *BFC*s with *BIC*s. In particular, the precision of mapping *BFC*s with *BIC*s in their case study was only 50% due to changes considered as bugs that, in fact, were not bug reports (e.g., feature request, refactoring). Furthermore, others authors highlighted limitations to map *BFC*s with *BIC*s due to some characteristics of the software that can negatively affect textual approaches. For example, German et al. investigated bugs that manifested themselves in unchanged parts of the software and their impact across the whole system (2009). Chen et al. studied the impact of dormant bugs (i.e., introduced in a version of the software system, but are not found until much later) on bug localization (2014). As opposed to the previous studies that have relied on the lines modified in the *BFC*s to identify the *BIC*, this study proposes (1) a model that helps researchers to reasoning whether the origin of a bug is intrinsic or extrinsic; and (2) how researchers can operationalize the model to identify the *BIC*, when it exists. Our preliminary approach (Rodríguez-Pérez et al. 2018b) was the seed to extend the work and provide a more comprehensive description of how to correctly identify *BIC*s.

Furthermore, in this work we detail the process of using the model and its operationalization to build reliable datasets that can be used to evaluate four existing SZZ-based algorithms.

4 The Framework and its Assumptions

Given a bug-fixing change (*BFC*), identifying its bug-introducing change (*BIC*) is not necessarily straightforward as bugs can have different origins as shown in Section 2. Thus, in order to identify when and how bugs were introduced, we designed a model that consists in a framework based on five assumptions. These assumptions enable the framework to describe the first time that the software exhibited the bug according to a *BFC*.

The model we propose is based on the following five assumptions:

1. The model assumes that there is **version control** for the software.
2. The model assumes that it has means to **identify the bug-fixing change (*BFC*)**.
3. The model assumes that it is possible to know **whether a bug is present** in the system or not.
4. The model assumes that it is possible to **identify a candidate of the bug-introducing change (*BIC*)** that corresponds to the bug-fixing change.
5. The model assumes that the **fix is perfect**.

The first assumption allows researchers to track how code changes as it evolves, and to recover any past version of it. The second one enables researchers to identify the *BFC*, and to link it to the contextual information of how the bug was fixed. The third assumption permits researchers to know when the software exhibited the bug that was fixed in the *BFC*. The fourth one allows researchers to identify whether the bug has been previously introduced in the SCM. And the fifth assumption enables researchers to decide that the bug is no longer present in the *BFC* snapshot, but it was present in a previous snapshot.

These assumptions can, to some extent, be implemented with today's technologies and processes. For some of them, however, we required theoretical conceptualizations and simplifications, as we discuss in an extensive way in the subsequent sections. We, therefore, offer details on how the model implemented each assumption. Furthermore, we inform researchers about known limitations and possible solutions for all assumptions. In those cases where an assumption, due to its theoretical or practical novelty, was elaborated more, we also provide context and introduce the necessary definitions and concepts.

4.1 The Model Assumes that there is Version Control for the Software

4.1.1 Implementation

The model assumes that the development history of the project is recorded in the source code management systems (SCM), and that the record is complete, i.e., it starts from the very first change[6] to the code. Thus, all changes can be tracked because they were done via a version control system (VCS) tool (such as `git`). For each change we can recover the

[6] A change is what developers do in a single commit

state of the system (i.e., snapshots of the system) before and after applying that change; and retrieve the differences between the two snapshots.

4.1.2 Limitations and Solutions

Nowadays, the history of a project is recorded in SCM, enabling researchers to reconstruct the process by which the software project was created (Bird et al. 2009). Although old software projects can migrate their history from previous repositories, the migration may not be complete (Gonzalez-Barahona et al. 2014). In addition, the use of SCM imposes some possible limitations that can alter how it was created. For example, changes may have been reordered, deleted or edited (Bird et al. 2009). In particular, commits in a pull-request might be reworked (in response to comments), and only those that are the result of the peer-review can be observed (Kalliamvakou et al. 2014). Another aspect to take into consideration is the effect of gatekeepers, who act as a filter/dispatcher for the incoming changes (Gousios et al. 2015; Canfora et al. 2011).

4.2 The Model Assumes that it has means to Identify the Bug-Fixing Change (*BFC*)

4.2.1 Implementation

When a bug report is closed by a *BFC*, the model assumes that it has means for linking the *BFC* with the bug report. If the system also uses a code review system, the model assumes there is a way to find the discussion corresponding to a given *BFC*. Therefore, a bug report can be linked to its *BFC* and the information related to its review.

4.2.2 Limitations and Solutions

Several studies that focus on issue tracker systems used to collect bug reports or feature requests have demonstrated that a substantial part of bug notifications are not correctly categorized, and are functionality requests or suggestions for refactoring. Herzig et al. reported 33.8% (2013), while Rodríguez-Pérez et al. reported up to 40% (2016). In addition, Herzig et al. pointed out that 39% of files marked as defective have never had a bug (2013).

Furthermore, when the bug notifications are correctly identified as a bug report, previous studies indicate several limitations of linking the *BFC* with the bug report. For example, the fixing commit cannot be linked to the bug (Bird et al. 2009), or the fixing commit was linked to a wrong bug report, as they do not correspond to each other (Bissyande et al. 2013).

A number of tools have been developed to increase the linkage between bugs and fixes, among others, EpiceaUntangler (Dias et al. 2015), BugTracking (Rodríguez-Pérez et al. 2016), Relink (Wu et al. 2011), Rclinker (Le et al. 2015), or Frlink (Sun et al. 2017). The model can use them in order to reduce these limitations, at least partially.

4.3 The Model Assumes that it is Possible to Know Whether a Bug is Present in the System or Not

4.3.1 Definitions and Concepts

To study the origin of bugs, our model needs to unequivocally determine if the bug is present for any given snapshot of the software system. In this way, we will be able to know when the bug appeared and when it has been fixed.

We need to consider what it means that “the bug is present”. Since there is no definition for ensuring that a bug is present in a snapshot, we build upon the definition of “defect” by IEEE Standard 1044 (2009):

“***Defect*****: An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced.**”

We will slightly adapt this definition in three ways: i) we will use the term “bug”, ii) we are only concerned with “software products”, and iii) we will add temporal behavior, by adding “at the moment of producing the snapshot”. The adapted definition will be as follows:

“***Bug*****: An imperfection or deficiency in a software product where that software product does not meet its requirements or specifications, as defined at the moment being considered, and needs to be either repaired or replaced.**”

Therefore, to know if a bug is present in a certain snapshot of the product, the model will check if it meets requirements or specifications at the moment of the production of the snapshot. This introduces an essential aspect as some lines of code might be considered a bug for a certain snapshot, because of the specifications at that point. However, the exact same lines could be considered correct if present in another snapshot if at that point some other specifications were applicable and were met (e.g., in Example 3 in Section 2).

As a result, we can define: A bug was present for the first time in the first snapshot where the fixed code can be considered incorrect in any branch that ends merged in the *BFC*’s branch, according to the specifications applicable to that snapshot. This definition considers that the bug can propagate several times, e.g., in multiple branches that lead to the *BFC*.

When developers fix a bug, they can write a test that fails if the bug is present (Beller et al. 2018). Thus, if developers could run such a test for every snapshot, they would see that the bug is not present in those snapshots where the test passes. We consider a test as *perfect*, if it can be run on any past version of the software.

This *perfect test* is a theoretical construct that may be challenging to create in practice. However, it provides an essential and precise definition of “faulty code at the time of writing it”. Furthermore, this *perfect test* can be seen as a kind of regression test[7] which will evolve and adapt depending on the software’s changing circumstance (e.g., dependencies, APIs, even requirements) for each past version. The perfect test would encompass all the knowledge about the behavior of the software in the past, thus forming an oracle for each previous version.

4.3.2 Implementation

Our model assumes that it is possible to know whether a bug is present in a system or not by using **perfect tests**. These tests would create a *signal* that pinpoints when the bug was present. For that, they can also be used with past snapshots, before the bug was fixed. Theoretically, these *perfect tests* would fail according to our previous definition[8].

[7] “regression testing is an activity aimed at showing that code has not been adversely affected by changes” (Rothermel and Harrold 1996)

[8] “the tests would fail in the first snapshot preceding the snapshot that fixed the considered bug, according to the specifications applicable for that first snapshot, i.e., for the requirements that were known and specified at that point”.

The idea of *perfect knowledge* replicates the idea of the *global observer* in distributed systems (Chandy and Lamport 1985); it is an **idealized situation** (i.e., difficult or even impossible to implement), but a beneficial concept for reasoning about the system, and for comparing practical implementations and algorithms.

In order to run the tests for previous snapshots, these tests might have to be updated "for past conditions", i.e., they have to be adapted to structural changes in the system under test (Moonen et al. 2008). In addition to the tested module, the tests need their dependencies: libraries, compilers or interpreters, external components and maybe even services accessed via remote APIs Zaidman et al. (2008, 2011). Thus, a test fails or passes not only for a certain snapshot, but for a certain snapshot of all those dependencies.

Dependencies can be considered as a part of the requirements (Mens et al. 2005): the module is expected to work, at any given moment, with a certain set of dependencies. Thus, the test should pass for that set. However, when dependencies change, the test may start failing, even if it is run on the same snapshot (Zaidman et al. 2011; Demeyer et al. 2002; Moonen et al. 2008; Marsavina et al. 2014; Palomba and Zaidman 2019; 2017). For example, the module can be expected to work with Python 2, but at some point the project decides that it should also run with Python 3. That will break large parts of the code, and many tests will fail when the new interpreter is introduced. Therefore, tests need to evolve to take into account the new dependency, in the same way they need to evolve to take into consideration any change in requirements.

Thus, the final definition of bug that we use in this work is:

"*Bug: An imperfection or deficiency in a software product that* ***causes a given test to fail****. The test will be defined for each snapshot of the product, according to the requirements and specifications applicable for that snapshot, and for the dependencies supported in it, and will fail for a snapshot only if the bug is present in that snapshot.*"

Although this definition may be difficult to implement in practice, it provides an accurate test to know when a bug is present, and therefore, when it is introduced. Assuming the model has perfect knowledge about the requirements, specifications, dependencies, and *perfect tests* are available, it can clearly describe when the bug is present, and from there on, it also knows when the bug was introduced, and how.

4.3.3 Limitations and Solutions

Being able to gather information of previous requirements, documentation or dependencies of a project in previous versions is not always easy, as shown by Zaidman et al. (2011). Some projects use build tools such as Maven or Gradle, and researchers can analyze the build scripts looking for dependencies or plugins that have changed. But, in other cases there is no formal record of such information. Thus, in the usual case a *perfect test* is not feasible. However, the contextual information found in issue tracker systems, code review systems and control version systems may help to write the tests, and to identify the origin of bugs.

Knauss et al. studied how the open communication paradigm in software ecosystems provides opportunities for 'just-in-time' requirement engineering (RE) (2014). They propose T-Reqs, a tool based on git that enables agile cross-functional teams to be aware of requirements at system level and allows them to efficiently propose updates to those requirements (Knauss et al. 2018). This tool can support successful implementation in our model, since researchers can match changes/updates in the requirements with the changes in the source code and then, our model can use this information to build the *perfect knowledge*.

4.4 The Model Assumes that it is Possible to Identify a Candidate of the Bug-Introducing Change (*BIC*) that Corresponds to the Bug-Fixing Changes

4.4.1 Implementation

To identify the *BIC*, the model assumes that there is a *perfect test* for the fixed bug. Any approach that uses the representation of the model should start by analyzing how to link the *BFC* to the contextual information of how the bug was fixed. Then, it can start looking for the corresponding *BIC*.

Finally, once the approach has the test for each snapshot, it runs the test for all the previous snapshots until it finds the first snapshot that fails according to a *BFC* or until the test cannot be run or build because the tested functionality is not implemented yet.

The theoretical possible outputs of the test are:

- The test passes for all snapshots. This means that the bug was never present until the *BFC*. This is impossible because if the test is perfect, that would mean there was no bug to fix. So, the model ignores this case.
- The test fails for at least some of the snapshots. This means that there will be a first snapshot for which the test fails. That snapshot will be the candidate *BIC*. It can be no other, because if the bug was in an earlier or later snapshot, the test would also fail for it.
- The test is not-runnable or not-building. The model does not consider these scenarios since it assumes that *perfect tests* can be updated to previous snapshots.

Once there is a candidate for the *BIC*, researchers can analyze why the test failed and determine whether this change introduce the bug of not:

- If there was no change in the source code that made the test fail, but the reason for the failure of the test was a change in requirements, specifications or dependencies, the candidate *BIC* is not responsible for introducing the bug. The change will be considered as the *FFC*. The model assumes that the bug is extrinsic because there is no new code causing the test to fail – the code introduced was correct (at least with respect to this bug).
- In any other case, the model assumes that the bug is intrinsic because the change includes code that causes the test to fail. Therefore, the candidate *BIC* is the *BIC*.

4.4.2 Limitations and Solutions

In practice, when manually inspecting the changes, we may not need *perfect knowledge*; we only need to be able to assert on whether the definition of a bug is fulfilled. We also need to consider that when we roll back into earlier snapshots, we could find a moment when the test cannot be run because the feature being tested was not implemented at that moment. Even in the presence of build automation tools such as Maven, it is sometimes not that easy to go back in time to rebuild a project (Zaidman et al. 2011). Moonen et al. have shown that about 2/3 of the refactoring changes from Fowler et al. (1999) can actually result in non-building test cases because the refactoring changes the original interface and the test code requires a change in the types of classes that were involved in the refactoring (Moonen et al. 2008). In contrast, Hilton et al. have recently performed a study on test coverage evolution using

Continuous Integration builds (Beller et al. 2017), reporting that this modern infrastructure eases building prior versions of a software project considerably (Hilton et al. 2018).

We could consider implementing these *perfect tests* by automatically generating them, e.g., using EvoSuite (Fraser and Arcuri 2013a; Palomba et al. 2016). However, automatically generating tests raises a number of issues. First, the generated test may not run or build in previous snapshots. Second, the test may not be precise enough since there will be lack of information to understand and implement the specifications and requirements. In fact, even if developers can implement the *perfect tests* manually because they have enough information, the results are not binary, as they might return four values: Pass, Fail, Not-Runnable and Not-Building. The test should return not-runnable when the feature to test is not present, and return not-building when there is an issue with the dependencies trying to be built in that snapshot (Zaidman et al. 2011; Moonen et al. 2008).

Nevertheless, researchers can use some test generation tools like EvoSuite (Fraser and Arcuri 2013a, b) to further investigate and solve theses issues. In particular, in future work we can investigate targeted search-based strategies to update tests after, e.g., refactoring operations (Vonken and Zaidman 2012).

Finally, another limitation is the assumption that the requirements in previous snapshots were always correct. If we combine that with the assumption that the tests are *perfect* and we can update them for conditions in the past, we run the risk of running into faulty requirements in previous snapshots (Viller et al. 1999). If we roll back the tests in this situation, the tests are likely to not fail.

4.5 The Model Assumes that the Fix is Perfect

4.5.1 Implementation

This means that the bug is no longer present after being fixed (i.e., after the *BFC*), and the bug report will not be reopened in the future. To ensure that the bug is no longer in the system, the model again uses the concept of *perfect tests*: if the snapshot of the *BFC* passes the test, the model ensures that, under the same specifications and requirements, the bug has been removed. We would then have what we call *perfect fixing*.

4.5.2 Limitations and Solutions

Perfect fixing is not always possible in practice and the bug report might need to be reopened (Zimmermann et al. 2012; Shihab et al. 2013).

In some cases, bug reports are reopened because they were not correctly fixed. Xia et al., reported that 6%-26% of the bug reports in Eclipse, Apache HTTP and OpenOffice.org were reopened. In this context, they proposed the ReopenPredictor tool which uses various kinds of features such as raw textual information or meta features to build a classification-based framework and predict whether a bug report would be reopened (2015).

Furthermore, Zimmermann et al. investigated the reasons why bug reports were reopened at Microsoft. Their findings showed that bug reports were typically reopened because either a tester did not provide enough information in the report and there was a misunderstanding about the cause of the bug, or the bug was a regression bug[9] (Zimmermann et al. 2012).

[9] “A regression bug is a bug which causes a feature that worked correctly to stop working after a certain event” (Brooks 1995)

4.6 Summary of the Assumptions

Table 4 summarizes the need, limitations and possible solutions for each assumption of the model.

5 The Model

In this section, we formally define the notions introduced in Section 2. We do this with two purposes in mind: (1) to identify the first manifestation of a bug in the history of a software product and, (2) to provide the formalisms used to create and describe a manually curated dataset which can be considered as the "ground truth". It is important to emphasize that the model is not a mathematical model solving relevant equations or characterizing the system, but it is a conceptual model that qualitatively represents the complex bug introduction process and highlights general rules and concepts. To that end, we use an example that identifies the bug-introducing change (*BIC*) or the first-failing change (*FFC*) given a bug-fixing change (*BFC*). This example describes a software product called Project A *(PA)* which uses an external library called *ExtL*. Figure 8 shows the model as a black box, with the information of a bug-fixing change as input and a change to the software identified as the bug-introducing change or the first-failing change as output.

5.1 Main Concepts & Unifying Terminology

We found that a unique terminology to name each of the concepts when identifying bug-introducing changes did not exist. We think that a common terminology would be desirable because researchers currently refer to different concepts as the same, and this can cause problems when trying to understand or reproduce previous studies. Table 5 offers a comparison of the terminology used in this work and how the concepts have been referred to in previous publications. To the best of our knowledge, no previous study has presented a comprehensive list of all these concepts and terms used, and neither has someone investigated whether the terms are being used consistently.

The terminology describes that developers using the source code management (SCM) to write software in terms of ***commits***, observable changes (additions, deletions or modifications) performed on a file (or set of files). The impact of a commit on a system might be represented as a ***snapshot***, which is a state of the project after the commit has been performed.

Depending on the origin of the bug, we distinguish between: an ***extrinsic bug*** which has its origin in a change not recorded in its source code,[10] or an ***intrinsic bug*** which has its origin in a change to the source code, this change is the bug-introducing change (*BIC*). Notice that extrinsic bugs do not have a bug-introducing change but a first-failing change (*FFC*).

To identify the bug-introducing change, we analyze the changes that fixed the bug in a bug-fixing change (*BFC*). To fix a bug, the bug-fixing change may add new lines or change (modify or delete) the existing ones. For a commit *c*, we label modified or deleted, but not added, lines as ***lines changed by a commit*** **LC(c)**.

If $LC(BFC) \neq \emptyset$, we can track down whether the revision which last modified each line in *LC(BFC)* lead to the bug that is fixed in the *BFC*, e.g., using tools such as "git blame". This last revision is called the ***previous commit*** *(pc)*.

[10] source code broadly defined as any file under version control

Table 4 Summary of the assumptions, their limitations and possible solutions

	Assumption	Need	Limitations	Solutions & tools
Section 4.1	Control version	VCS	History incomplete (Gonzalez-Barahona et al. 2014)	–
			Changes in commits (Kalliamvakou et al. 2014)	–
			Gatekeepers' Effect (Canfora et al. 2011)	–
Section 4.2	Identify *BFC*	Bugs reported and fixed	Bugs vs. features (Herzig et al. 2013)	BugTracking (Rodríguez-Pérez et al. 2016)
			Linkage (Rodríguez-Pérez et al. 2018a; Bird et al. 2009)	Relink (Wu et al. 2011), Rclinker (Le et al. 2015), Frlink (Sun et al. 2017)
			Tangled commits (Rodríguez-Pérez et al. 2018a)	EpiceaUntangler (Dias et al. 2015)
Section 4.3	Presence of a bug	Perfect test	Bug dependent on specifications	Improving RE (Knauss et al. 2014)
			(Zaidman et al. 2011; Moonen et al. 2008; Viller et al. 1999)	T-Reqs (Knauss et al. 2018)
Section 4.4	Identify *BIC*	Perfect test signals bug	Test not building/running (Zaidman et al. 2011; Moonen et al. 2008)	EvoSuite (Fraser and Arcuri 2013b; 2013a)
			Faulty requirements (Viller et al. 1999)	T-Reqs (Knauss et al. 2018)
Section 4.5	Perfect Fixing	*BFC* fixes bug definitely	Bug report reopened (Zimmermann et al. 2012; Shihab et al. 2013)	ReopenPredictor (Xia et al. 2015)

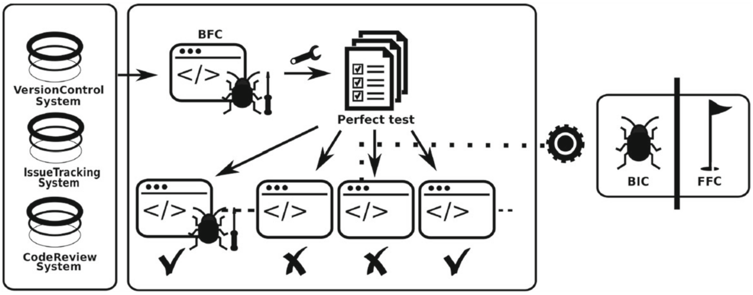


Fig. 8 Model to identify bug-introducing changes (*BIC*s) or first-failing changes (*FFCs*)

Since the bug-fixing change can change more than one line, it is possible that different lines in *LC(BFC)* may have different previous commits. We will refer to *PC(c)* as the set of previous commits of a commit.

But, it is also possible to go further back in time and recursively analyze the previous commits of the *LC(pc)*. These commits are referred to as ***descendants commits*** of a bug-fixing change, *(DC(BFC))*. The previous commits are the immediately previous commits to the lines changed in the bug-fixing change; the descendant commits are all the commits that previously modified the lines changed in the bug-fixing change. The remaining commits in the source code management of a software product from the bug-fixing change backwards are the ***ancestors commits***, $AC(BFC)$, which also includes the previous and descendants commits. Formally,

$PC(BFC) \cup DC(BFC) \subseteq AC(BFC)$.

5.2 A Process to Identify when and How a Bug was Introduced

This subsection describes the process used by our proposed model (Section 4) to determine when and how a bug was introduced. This process can be generalized and allows us to demonstrate how existing SZZ-based algorithms can be evaluated, which is something missing in the current literature.

This process consists of the following steps, which can be adopted by other researchers as well.

Ensure that a Control Version Exists The first step is to ensure that the selected project has a development history recorded in a SCM. Furthermore, to identify every change in the code from the beginning of the project until the bug fixing change, we need to ensure that the SCM of the selected project holds the complete history of the project.

Identify the Bug-Fixing Change (*BFC*) The second step is to identify the bug-fixing change linked to a bug report. To that end, researchers should analyze only issues labeled (manually or by developers) as bugs reports.

When analyzing a bug fix, it is important to consider that a *BFC* may fix different bugs; and that fixing a bug might require multiple partial fixes (commits). Furthermore, a *BFC* can modify other parts of the source code that are not related to the bug, e.g., removing dead code or refactoring the source code (Rodríguez-Pérez et al. 2018a; Neto et al. 2018). Thus, when those cases exist, researchers should only analyze the source code of the *BFC* that fixed the aimed bug.

Table 5 Comparison of our terminology with the one found in the research literature

Proposed Terminology	Found as...	References
Commit	Change	(da Costa et al. 2017; Kim et al. 2006)
	Commit	(Izquierdo-Cortazar et al. 2011; Izquierdo-Cortázar et al. 2012)
	Revision	(Kim et al. 2008)
	Transaction	(Śliwerski et al. 2005; Bettenburg and Hassan 2013; Kim et al. 2006)
Previous commit	Earlier change	(Śliwerski et al. 2005)
	Change immediately prior	(Williams and Spacco 2008)
	Last change	(da Costa et al. 2017; Bavota and Russo 2015)
	Previous commit	(Izquierdo-Cortazar et al. 2011)
	Recent version	(Kim et al. 2008)
	Preceding revision	(Hata et al. 2010; Rahman et al. 2011; Guerrouj et al. 2015)
Descendant commit	Descendant	(Alexandru and Gall 2015)
	Change	(Śliwerski et al. 2005; da Costa et al. 2017; Kim et al. 2006)
	Commit	(Kim et al. 2008)
Ancestor commit	Revision	(Śliwerski et al. 2005; da Costa et al. 2017; Kim et al. 2006)
	Change	(Kim et al. 2008)
	Commit history	(Meneely et al. 2013)
Bug-fixing change	Fix for a bug	(Śliwerski et al. 2005)
	Bug-fixing change	(da Costa et al. 2017; Kim et al. 2006; Williams and Spacco 2008; Izquierdo-Cortazar et al. 2011; Kim et al. 2008)
	Fixed revision	(Hata et al. 2010)(Pan et al. 2009)
Bug-introducing change	Fix-inducing changes	(Śliwerski et al. 2005; Williams and Spacco 2008; Izquierdo-Cortázar et al. 2012)
	Bug-introducing change	(da Costa et al. 2017; Kim et al. 2006; Kim et al. 2008)
	Defect-inducing	(Syer et al. 2015)
First-failing change	Fix-inducing changes	(Śliwerski et al. 2005; Williams and Spacco 2008; Izquierdo-Cortázar et al. 2012)
	Bug-introducing change	(da Costa et al. 2017; Kim et al. 2006; Kim et al. 2008)
	Defect-inducing	(Syer et al. 2015)

Ensure the Perfect Fixing The third step is to ensure that the perfect fixing exists. A *BFC* might be incomplete and spread over several commits. In such cases, there is no *perfect fixing*. However, researchers need to be sure of this fact when analyzing the origin of bugs and they have to identify whether a bug report was reopened or not. In the affirmative case, researchers should consider the last *BFC*.

Describe Whether a Bug is Present The fourth step is to describe whether a bug was present in a certain snapshot or not. For that, researchers can use all the information available in the SCM, in the ITS, in the code review system and/or in the testing system to build the ***perfect test*** signaling a bug, as explained in Section 4.3.

Thus, in order to describe whether a certain snapshot contains the bug fixed in the bug-fixing change, researchers need to run the perfect test from the bug-fixing snapshot backward. If the test passes, the snapshot does not contain the bug but, if the test fails, the snapshot contains the bug.

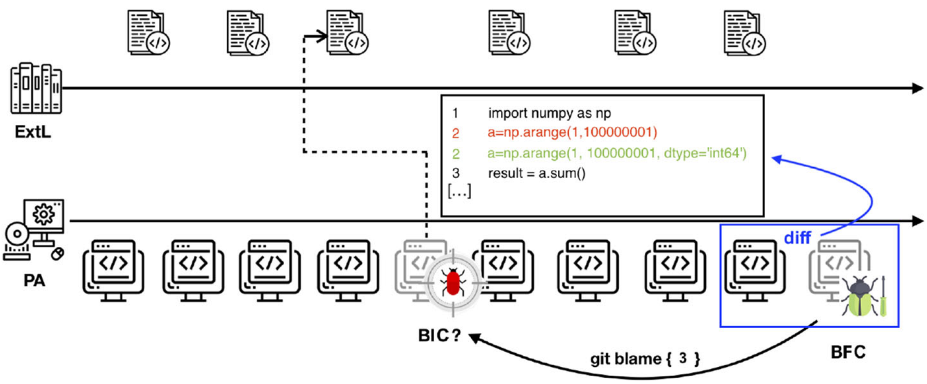
Identify the First-Failing Change The last step is to identify the first-failing change given a bug-fixing change and decide whether it is the bug-introducing change or not. To find the first-failing change, we assume linear history and need to identify the first snapshot in the continuous sequence of test failing snapshots, which finishes right before the bug-fixing change. That is, there is a continuous sequence of snapshots for which the test fails, starting in the possible first-failing change, and finishing right before the bug-fixing change. Since the test is failing –all the way– from this snapshot up to the fix, we can say that this is the first snapshot "with the bug present", thereby we have identified the first-failing change. Furthermore, if this change introduced the bug, it is the bug-introducing change.

We use the example in Fig. 9 to illustrate how researchers can distinguish both scenarios. Figure 9 shows the timeline of Project A (PA) represented by its snapshots from the bug-fixing change backward, and the timeline of an external library (ExtL) used in PA. The following scenarios are possible when analyzing the first snapshot in the continuous sequence of test failing snapshots:

- The bug is intrinsic. The *LC(commit)* introduced the bug because the lines were faulty. For example, Fig. 9 shows how line 2 added in the previous commit of bug-fixing change inserted the bug. This line uses an external library (*numpy*) in a wrong way causing the bug to appear and manifest itself for the first time in the bug-introducing change. In this case[11], the documentation of *numpy* clearly describes that by default "arange" infers the data type from the input, thereby the line uses *numpy* in a wrong way causing the bug. This snapshot is the bug-introducing change.
- The bug is extrinsic. The *LC(commit)* did not introduce the bug. For example, Fig. 10 shows how line 3 inserted in a previous commit of the bug-fixing change did not insert the bug because these lines are using ExtL, which contained a bug. In this case,[12] the method $array.split()$ returns an incorrect behavior with array size bigger than *MAX_INT32*. This snapshot is not the bug-introducing change, but the first-failing change.

[11] https://stackoverflow.com/questions/43209391/numpy-is-calculating-wrong

[12] https://github.com/numpy/numpy/issues/11809



```
1   import numpy as np
2   a=np.arange(1,100000001)
2   a=np.arange(1, 100000001, dtype='int64')
3   result = a.sum()
[...]
```

Fig. 9 Guiding example to identify how the bug was inserted given a bug-fixing change (*BFC*) in Project A (PA)

6 Operationalizing the Process

This section details how we operationalized the process described in Section 5.2. This operationalization is essential to identify the origin of bugs in real open source projects because the model (Section 5) is based on five idealized assumptions (Section 4).

Ensure that a Control Version Exists The projects that we selected have a development history recorded in a SCM. Also, for both projects, the initial commit[13] was not migrating code from other version control system. Thus, we were able to trace back all the development history of the projects without suffering from the initial import commits observed by Da Costa et al. (da Costa et al. 2017).

Identify the Bug-Fixing Change To identify the *BFC*, we only analyzed issues labeled (manually or by developers) as bugs. Then, from these bugs, we excluded the bugs where developers do not agree whether the *BFC* was fixing the bug or another kind of issue (Herzig et al. 2013). In total, we discarded four *BFCs* (see Section 7.1).

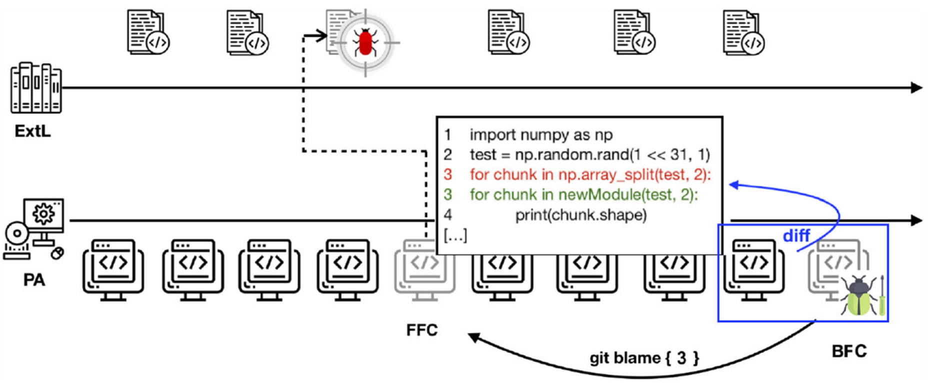
When analyzing a bug fix, we were aware that a *BFC* might (1) fix different bugs; (2) require multiple partial fixes (commits); and (3) modify other parts of the source code unrelated to the bug. When we identified these cases, we only analyzed the source code of the *BFC* that fixed the aimed bug.

Ensure the Perfect Fixing When we found reopened bug reports, we selected and analyzed the last *BFC* identified in the ITS. In total, we found two cases of reopened bug reports.

Although we analyzed bug reports listed until 2016, we cannot assure that these bug reports will not be reopened in the future. However, if these bug reports have not been reopened for at least two years, we can be almost sure that the *BFC*, indeed, fixed the bug.

Describe Whether a Bug is Present Ideally, we should contact developers to identify whether a bug was present in a certain snapshot or not because they are the project experts.

[13] https://github.com/elastic/elasticsearch/commit/b3337c; https://github.com/openstack/nova/commit/bf6e6e



```
1   import numpy as np
2   test = np.random.rand(1 << 31, 1)
3   for chunk in np.array_split(test, 2):
3   for chunk in newModule(test, 2):
4           print(chunk.shape)
[...]
```

Fig. 10 Guiding example to identify how the bug was inserted given a bug-fixing change (*BFC*) in Project A (PA). In this scenario the bug was extrinsic, caused by a bug the External Library (ExtL) that PA is using. It manifested itself in the lines inserted in the first-failing change (*FFC*)

However, in practice, this is hard to implement because developers' time is limited. Furthermore, even if developers participate, they might not be able to decide whether a specific snapshot did not introduce the bug because it fulfilled the requirements of the project in previous snapshots. Indeed, developers might have forgotten those requirements, might misinterpret them retrospectively or might not even have been involved in the project at that time (da Costa et al. 2017).

Thus, we have to trust the knowledge of researchers, in this case, the authors of the paper. Although we are not experts developers in Nova and ElasticSearch, we had information in the ITS, the source code review system and the SCM that, when analyzed, helped us to identify whether a bug was present in a snapshot. To describe whether a bug was present in a snapshot, we needed to build the "perfect tests": however, there are no practical means to implement and run the perfect test. Thus, we mentally created and ran the designed test on the previous snapshots and reasoned whether we could assert that these snapshots fulfilled the requirements of the project. We used this mentally designed test as proxy of the "perfect test".

For example, a valuable piece of information to mentally create the "perfect test" was the description[14] of the bug report #1410622 in Nova. This description suggests that this is an extrinsic bug as its origin was a change in an external library (which is not recorded in the SCM of Nova). Other useful information came from the comments and discussion from developers in the ITS. A developer's comment[15] at Nova bug #1370590 indicates that the bug is extrinsic because the bug has its origin in a requirement change. A condition was introduced during development that needed some information, but many calls to a function were not providing this information since it was not required before which caused the bug.

Identify the First-Failing Change We classified the bug as intrinsic or extrinsic after identifying the *BFC* and using the mentally designed test as proxy of the "perfect test".

[14] The description was: *"Webob library has a bug Pylons/webob#149 which causes modification of req.body after first access. So it's critical to calculate the body hash before any other access is made."*

[15] The comment said: *"These calls now need to provide disk_info to _create_domain_and_network"*

For the extrinsic bugs, we linked their *BFC* with the presence of a *FFC* because no *BIC* can be found in the SCM. For the intrinsic bugs, we applied *"git diff"* to the files touched by the *BFC*. *"Git diff"* identified what lines were added, modified or deleted between the snapshot after the *BFC* and the previous one. That way, we determined the previous commits of the *BFC* (*PC(BFC)*) and analyzed whether these previous commits introduced the bug. Notice that, the lines that did not contain source code (e.g., comments or blank lines) or affect test files were filtered out. *"Git diff"* cannot backtrack the lines that have been added. Thus, when we identified a *BFC* with only new lines added to fix the bug, we analyzed the lines adjacent to these added lines. This analysis provides good perspective to understand whether the last modification of these adjacent lines were somehow faulty, e.g., the adjacent lines were missing a piece of code to function correctly (i.e., the lines added by the *BFC*).

Then, we selected each one of the previous commits to analyze whether the test would fail in the corresponding snapshot. If the test would not fail in none of the snapshots from previous commits, we navigated back to the previous ones, the descendant commits, until finding the first commit for which the test would fail; this commit was the *BIC*. Due to the complexity of some bugs, sometimes we could not manually identify the *BIC*.

At the end of this process, we could have following three different outcomes:

- The *BFC* had a *BIC*, and we identified it manually.
- The *BFC* had a *BIC*, but we did not identify it manually.
- The *BFC* did not have a *BIC*, but a *FFC*.

7 Case Studies

Following Easterbrook et al. (2008), we selected two exploratory case studies to gain a deep understanding of the bug introduction phenomenon in two open source projects: Nova and ElasticSearch. We applied the model proposed in this paper to both projects to evaluate its applicability and to provide important insights that led us to validate or refute the assumption that relates the lines fixed in the *BFC* to the lines that introduced the bug in the software product. The operationalization of the model in these case studies resulted in a procedure that allowed us to build curated reference datasets in which bugs are linked with the presence or absence of *BIC*s. These manually curated datasets can be considered as the ground truth of the projects computing the real performance of the algorithms that are build upon the current assumption.

Runeson et al. argued that the systems selected in a case of study must be typical in order to generate a theory based on them (Runeson et al. 2012). Our aim is not to generate a theory of bug introduction, but to qualitatively study the bug introduction phenomenon in open source projects. Thus, we selected two projects with interesting and worthwhile characteristics to study. Besides, both projects have some differences that can allow us to validate the model and may extend the procedure to other similar open source projects. The second case study can be seen as an analytical replication of the first one.

Nova is the most active module of the OpenStack project in terms of contributions. OpenStack has more than 7,900 contributors, and significant industrial support from several major companies such as Red Hat, Huawei, IBM or HP. Nova is mainly written in Python and currently has more than 52,600 commits with more than 500K lines of code and around

1500 developers.[16] All its history is saved and available in a version control system (git[17]), an issue tracker system (Launchpad)[18] and a source code review system (Gerrit[19]).

In addition to the enormous diversity of people and companies contributing to Nova, the project has other characteristics that make it a good case to study: (1) the ease of gathering data. An important factor to ensure the reliability of data is that in a previous study (Rodríguez-Pérez et al. 2016) we had already identified bug reports in the issue tracker system; (2) Nova uses Python, a dynamically typed, interpreted programming language. Python is dynamically typed and this can affect the way that bugs were introduced into the source code of a project. Ray et al. claim that statically typed languages are less defect prone than the dynamic typed languages (Ray et al. 2014), although there is some controversy about this work (Berger et al. 2019); and (3) it uses a source code review system that it is connected with a continuous integration (CI) tool in order to verify that quality criteria are satisfied before a code change is integrated in the repository (Vassallo et al. 2016).

ElasticSearch is a distributed open source search and analytics engine written in Java (a statically typed language). It has over 30,500 commits and over 900 developers, which points towards a frequent evolution in the code. This project was chosen because of its rigorous policy of labeling issues, as ElasticSearch developers use the label "bug" to tag issues that describe real bugs. We can thus be sure that the *BFC*s address real bugs. The code and the bug report list of ElasticSearch are hosted on GitHub.[20]

In addition, ElasticSearch has other characteristics that makes it a good case study: (1) the ease of gathering the data since its code is hosted on GitHub. In addition, the policy of adding the link of the bug report number or the pull request number into the *BFC* is helpful when linking and analyzing the two data sources; (2) It is a statically-typed language project written in Java and this programming language might present different characteristics than Python; (3) It uses a source code review that is built into the pull requests system of GitHub. That way, reviewers can discuss and review the proposed changes and add follow-up commits before these changes are merged into the base branch of the project.

7.1 Nova and Elasticsearch Datasets

We relied on the Nova dataset from our previous work (Rodríguez-Pérez et al. 2016). This dataset consists of 60 random bug reports that were reported in 2015. For each of the bug reports that we manually identified, two different researchers manually linked them to the *BFC*s in the SCM. Then in ElasticSearch, we randomly gather 60 fixed and closed issues labeled as bug and reported between January 2013 and December 2016 from the GitHub issue tracker. Subsequently, we manually checked that the *BFC* was correctly linked.

To ensure that the bug reports could be applied to our model, we verified that they describe *real* bug reports at the moment of their report and not other issues (as the ones studied by Herzig et al. (2013)). For that, we carefully read the description and comments in the issue tracker system and code review system to analyze whether we could apply the model.

[16] http://stackalytics.com

[17] https://wiki.openstack.org/wiki/Getting_The_Code

[18] https://launchpad.net/openstack

[19] https://review.openstack.org/

[20] https://github.com/elastic/elasticsearch/

For example, the description of the bug report #1185290[21] looked like a bug. However, after carefully analyzing all the comments, this report was removed because of the discordance between developers of Nova:

- "I am not sure that I consider this a bug. Without –all-tenants=1, the code operates under your own tenant. That means that –all-tenants=1 foo should really be a no-op without –all-tenants=1."
- "I disagree, mainly because the structure of the requests and code path should largely be transparent to the user. I would suggest that specifying –tenants should imply you are doing a query across –all-tenants=1unless the –tenants specified is the same as what is contained in OS_TENANT_NAME (the unless part is debatable)"

Furthermore, we uncover other reasons why some bug reports cannot be applied to our model. For instance, bug report #1431571[22] described a bug in a test file. We removed it because a bug in a test file does not mean that the source code of the project contained a bug. We also discovered that bug reports such as #7740[23] and #1448075[24] described hypothetical scenarios (i.e., a possible bug in the future). These bug reports were excluded from the analysis because, although developers described them as bug reports, the bug was still hypothetical and had not occurred yet in the project. As such, we could not build the *perfect test* in those cases as the *BFC*s were fixing hypothetical future bugs.

The result of this analysis was the removal of 3 bug reports from the initial set of 60 random bug reports in Nova and of a 1 bug report from the initial set of 60 random bug reports of ElasticSearch.

7.2 Results

This section answers the research questions. First, we present the model that helped to describe when a snapshot of a component exhibits the bug. Then, we describe the empirical results of the evaluation of this model. We applied the model on two different datasets, from Nova and ElasticSearch, with the aim of identifying intrinsic and extrinsic bugs. In this process, we obtained curated and reliable datasets in which each *BFC* was connected to a *BIC* or a *FFC*. Finally, we used these curated datasets to compute the effectiveness of four existing SZZ algorithms.

7.2.1 RQ1: Can there be Criteria to Help Researchers Find a Useful Classification of Changes Leading to Bugs?

To better understand and solve the problem of identifying the origin of bugs, we designed a model (Section 5) that provides criteria (Section 4) for reasoning which snapshot of a software product first exhibited the bug. Specifically, the model is based on the idea of the "perfect test" which was designed using prior literature (Rodríguez-Pérez et al., 2018a, b; Kim et al., 2006; Sinha et al., 2010) and empirical findings (da Costa et al., 2017; Rodríguez-Pérez et al., 2018a, b).

[21] https://bugs.launchpad.net/nova/+bug/1185290

[22] https://bugs.launchpad.net/nova/+bug/1431571

[23] https://github.com/elastic/elasticsearch/issues/7740

[24] https://bugs.launchpad.net/nova/+bug/1448075

To ensure that the criteria defined in the model, in particular the “perfect test”, can be applied to real-world projects, we manually analyzed the origin of the 116 bugs in two open source projects. For that, we applied the operationalization of the process as is described in Section 6 into the two projects and then, we evaluated whether the criteria helped us to find a useful classification of the origin of bugs.

This study shows that, contrary to what is assumed in the literature (i.e., the last change that touched the fixed line(s) in a bug-fixing change introduced the bug), there are other sources for the introduction of bugs (e.g., changes in external dependencies, or changes in requirements). Although these sources were already known, our proposed model is the first one that includes them as a part of the model.

Furthermore, this careful analysis enabled us to produce manually curated datasets for Nova and ElasticSearch with bug-introducing changes and bugs that were not introduced by any change in the source code. Thus, we classified bugs as intrinsic and extrinsic and calculated the share of *BFC*s that have and do not have a *BIC*.

We determined whether the bug was intrinsic or extrinsic by applying our criteria to the projects. Although, by definition, intrinsic bugs always have a *BIC*, sometimes, we were unable to identify it manually. The complexity of the source code and the lack of information, when we analyzed the *BFC*s, made this identification difficult because we could not (mentally) implement the test. Thus, our results have intrinsic bugs with and without a *BIC* found. Table 6 shows the number of intrinsic bugs for which the *BIC* was manually found (or not), the number of extrinsic bugs having a *FFC*, and the number of bugs that we could not be sure whether they were intrinsic or extrinsic.

Notice that classifying a bug as “*BIC* not found” is different from not having a *BIC*. When we were sure that there was no *BIC* causing the test to fail, we classified the bug as not having a *BIC* (extrinsic). However, we classified the bug as “*BIC* not found” when we were sure that a *BIC* exists (the bug was intrinsic), but we were unable to find this *BIC* manually.

We observe that the lion’s share of *BFC*s, both in Nova(60%) and in ElasticSearch(64%), were related to intrinsic bugs, previous changes or omissions caused these bugs. The percentage of extrinsic bugs is higher in Nova (21%) than in ElasticSearch (9%).

RQ1: The criteria defined in the model enables us to classify bugs as intrinsic or extrinsic and to create manually curated datasets that contain information about intrinsic (with the ID of the *BIC*) and extrinsic bugs. In our case studies, 9%–21% of the bugs were extrinsic, meaning that they do not have a *BIC* in the SCM.

7.2.2 RQ2: Do These Criteria Help in Defining Precision and Recall in Four Existing SZZ-Based Algorithms for Detecting Bug-Introducing Changes?

After the positive answer for RQ1, we obtained the manually curated datasets that can be understood as the“ground truth” datasets. We applied four existing SZZ algorithms that

Table 6 Percentage of intrinsic bugs with a bug-introducing change (*BIC*) manually found (or not); extrinsic bugs with a first-failing change (*FFC*); and undecided bugs in Nova and ElasticSearch (*ES*)

	Intrinsic		Extrinsic		
	BIC found	BIC not found	FFC	Unsure	Total
Nova	34 (60%)	4 (7%)	12 (21%)	7 (12%)	57
ES	38 (64%)	7 (12%)	5 (9%)	9 (15%)	59

retrieve the *BIC* given the *BFC* to our datasets. In particular, we used (1) the original SZZ (Śliwerski et al. 2005), this algorithm links the SCM and the ITS in order to identify the *BFC* and then, it identifies a set of changes that, according to the algorithm, are flagged as the *BIC(s)*. This set is identified by determining the lines that have been changed between the *BFC* and its previous version (e.g., diff) and identifying the last change(s) to those lines (e.g., git blame). Then, it uses a temporary window from the bug report date until the *BFC* date to remove some false positives from the set of previous changes. The remaining changes in this set were blamed as the bug-introducing change(s) by the SZZ algorithm. We used (2) the SZZ-1 (Kim et al. 2006), this algorithm is an improvement of the SZZ algorithm, which uses annotation graphs to reduce false positives and gain precision by excluding comments, blank lines, and format changes from the analysis. We use the SZZ-1 with two different heuristics; (3) SZZ-1E (Izquierdo-Cortazar et al. 2011) that identifies an unique *BIC* as the earlier commit from the set of *PC(b)* and, (4) SZZ-1L (Davies et al. 2014) that identifies an unique *BIC* as the latest commit from the set of *PC(b)*. The four SZZ approaches do not attempt to identify *FFCs* since they do not consider that a bug can be caused by change(s) not recorded in the SCM. For that reason, all the previous changes identified by the SZZ are considered to be *BIC*s.

After that, we compared the manually curated datasets with the results from the four existing SZZ algorithm and measured how many *BIC*s (true positives) these algorithms obtain, how many identified commits were not the *BIC*s (false positives), and how many *BIC*s could not be found (false negatives).

Our criteria helps to determine the first snapshot of a software component that exhibits the bug according to a bug-fixing commit and identify the bug-introducing change. However, according to our model, there is just one change that introduced the bug. Notice that, because of the heuristics of SZZ and SZZ-1, there can be more than one *BIC* for a *BFC*. Thus, we can have a set of previous commits (*PC(BFC)*) identified as *BIC*s by SZZ and SZZ-1. To compare our manual curated datasets with these algorithms' results and evaluate their performance, we counted the number of true positives, false positives, and false negatives using the following criteria:

1. **ALL**: We counted all the commits that the algorithms identified for a bug-fixing commit as true positives or false positives. When $|PC(BFC)| > 1$; we counted one true positive whether the *BIC* existed and it belonged to the set of *PC(BFC)*. We flagged as false positives the other changes belonging to the set of *PC(BFC)*. For example, when we applied SZZ and SZZ-1 to #1486541 of Nova these algorithms identified three *BIC*s and the set of previous commits was three ($PC(BIC) = 3$). But, just one of these previous commits was the change that introduced the bug reported in #1486541 (there is one true *BIC*). Thus, we identified one true positive and two false positives for the *BFC* that fixed #1486541. When none of the changes in the set of *PC(BFC)* was the *BIC*, we counted all of them as false positives. In case the algorithms could not be applied (i.e., *BFC*s with only new lines added to fix the bug) but there was a change that introduced the bug, we counted one false negative.
2. **At least**: We counted a true positive whether there was a *BIC* and it was in the set of previous commits (*PC(BFC)*) identified by the algorithms. For example, in #1486541 of Nova $PC(BFC) = 3$. Although just one of the previous commits was the change that introduced the bug (*BIC*), we counted one true positive and zero false positives for the *BFC* that fixed #1486541. When none of the changes from the *PC(BFC)* introduced the bug, we counted one false positive. In case the algorithms could not be applied but

Table 7 Nova project: Results of True Positives *TP*, False Positives *FP*, False Negatives *FN*, Recall and Precision for SZZ-based algorithms assuming that: (1) SZZ and SZZ-1 flag all of the commits belong to a set of *PC(b)* as *BIC*; and (2) the four existing SZZ algorithms only flag the earlier *SZZ-1E* or latest *SZZ-1L* commits that belongs to a set of *PC(b)* as *BIC*

	SZZ			SZZ-1				
	All	At least	Only	All	At least	Only	SZZ-1E	SZZ-1L
TP	25(29%)	25(54%)	17(37%)	28(33%)	28(61%)	19(41%)	25(52%)	19(36%)
FP	54(61%)	12(26%)	20(43%)	51(59%)	11(24%)	20(43%)	14(29%)	20(37%)
FN	9 (10%)	9(20%)	9(20%)	7(8%)	7(15%)	7(15%)	9(19%)	15(27%)
Precision	0.32	0.68	0.46	0.35	0.71	0.48	0.64	0.49
Recall	0.74	0.74	0.65	0.80	0.80	0.73	0.73	0.56
F-score	0.44	0.70	0.54	0.49	0.76	0.58	0.68	0.52

there was a change that introduced the bug reported in the bug report, we counted it as one false negative.

3. **Only**: We counted a true positive whether there was a *BIC*, it was in the set of previous commits (*PC(BFC)*) identified by the algorithms and $PC(BFC) = 1$. For example, in #1486541 of Nova $PC(BFC) = 3$, although one of them was the *BIC* we counted one false positive because $PC(BFC) > 1$. In case that the algorithms could not be applied but there was a change that introduced the bug reported in the bug report, we counted it as one false negative.

When we applied the SZZ and SZZ-1 algorithms to the set of 46 *BFC*s of Nova,[25] we obtained 79 changes considered as *BIC*s by the algorithms. When these algorithms were applied to the 43 *BFC*s of ElasticSearch,[26] we obtained 85 changes flagged as *BIC*s. On the contrary, when we applied SZZ-1E and SZZ-1L to the 46 *BFC*s of Nova and 43 *BFC*s of ElasticSearch, these algorithms returned 43 and 36 changes flagged as *BIC*s respectively.

Tables 7 and 8 present the percentage of true positives (TP), false positives (FP) and false negatives (FN), the precision ($Precision = \frac{TP}{TP+FP}$), recall ($Recall = \frac{TP}{TP+FN}$) and F-Score ($F\text{-}Score = 2\frac{(Precision * Recall)}{Precision + Recall}$) of the SZZ algorithms. There are no True Negatives (TN) in the tables because these would be commits that did not introduce the bug and were not identified by the algorithms, i.e., all the ancestor commits that SZZ does not identify.

When comparing the number of true positives from the SZZ approaches, we observed that the assumption "a bug was introduced by the lines of code that were modified to fix it" varies depending on the approach and the criteria being used. For example, Tables 7 and 8 shows that this assumption holds better in SZZ and SZZ-1 (54%-63%) when we consider that at least one of the changes identified by the algorithm is the *BIC*. The other results showed that the assumption holds in less than a half of the bugs analyzed in both projects.

Tables 7 and 8 show that in both projects, the highest precision, recall and F-Score were obtained using the SZZ-1 algorithm and the "At Least" evaluation criteria. Furthermore, from these Tables we see that the SZZ-1 performed sightly better than the original SZZ algorithm.

[25] Out of the 46 bugs, we manually found 34 *BIC*s and 12 *FFCs*. We removed one bug because we were unsure about its origin.

[26] Out of the 43 bugs, we manually found 36 *BIC*s and 5 *FFCs*. We removed three bugs because we were unsure about their origin.

Table 8 ElasticSearch project: Results of True Positives *TP*, False Positives *FP*, False Negatives *FN*, Recall and Precision for SZZ-based algorithms assuming that: (1) SZZ and SZZ-1 flag all of the commits belong to a set of *PC(b)* as *BIC*; and (2) the four existing SZZ algorithms only flag the earlier *SZZ-1E* or latest *SZZ-1L* commits that belongs to a set of *PC(b)* as *BIC*

	SZZ			SZZ-1				
	All	At least	Only	All	At least	Only	SZZ-1E	SZZ-1L
TP	26 (27%)	26 (61%)	12(28%)	27 (27%)	27(63%)	12(28%)	16 (28%)	19 (35%)
FP	59 (61%)	10 (23%)	24(56%)	58 (61%)	9(21%)	24(56%)	20 (34%)	17 (30%)
FN	12 (12%)	7(16%)	7(16%)	11 (12%)	7(16%)	7(16%)	22 (38%)	19 (35%)
Precision	0.31	0.72	0.33	0.32	0.75	0.33	0.44	0.53
Recall	0.68	0.79	0.63	0.71	0.79	0.63	0.42	0.50
F-score	0.42	0.75	0.44	0.44	0.77	0.43	0.43	0.51

The real performance of the four existing SZZ algorithms showed that although the most effective results were obtained by SZZ-1 and the "At Least" criteria, the four algorithms reached a low percentage of true positives, in which the best case was 61% in Nova and 63% in ElasticSearch.

RQ2: SZZ-1 performed better than the original SZZ in both projects. SZZ-1 obtained the best performance when applying the "At Least" criteria. However, the percentage of true positives was relatively small when this criteria was not used $\sim$ 38% in Nova and $\sim$ 30% in ElasticSearch.

What Causes a Previous Commit (as Identified by SZZ-Based Algorithms) to not be the *BIC*? In those cases where the previous commit identified by the four existing SZZ algorithm was not the *BIC*, we investigated the cause for the misclassification. Some of these reasons are already known from previous studies and, although we do not pretend to do an exhaustive classification of why the previous commits analyzed were not the *BIC*, we identified some other reasons that have not been taken into account previously and added them to the next list of reasons:

- **The bug was already in the modified line** (da Costa et al. 2017; Neto et al. 2018; Kim et al. 2006): The modified line was buggy, but the bug was introduced before the last modification. For instance, one of the previous commits (*57108c8575b*) of bug[27] #4564 fixed another bug. But lines 194–195 modified by this previous commit already contained the buggy code that caused bug #4564. Thus, this previous commit did not introduce bug #4564, but it was introduced by a descendant commit of these buggy lines 194–195.
- **The *BIC* was not in the *DC(b)* or *AC(b)* because it was an extrinsic bug**: The modified line has never been buggy from its introduction. For instance, the bug caused by a change in an external artifact[28] #3551 explained in the Fig. 3, *Example 1* in Section 2.
- **The *BFC* only added new lines to fix the bug** (da Costa et al. 2017; Kim et al. 2006;

[27] https://github.com/elastic/elasticsearch/issues/4564

[28] https://bugs.launchpad.net/nova/+bug/1449028

Rodríguez-Pérez et al. 2018b): Due to how SZZ works, it cannot identify the case with only new lines in the *BFC*. For instance, commit *2442e1fb* forgot to add an *if* condition. Thus, the *BFC*[29] for bug #2566 only added new lines to fix the bug.
- **The previous commit made an equivalent change in line(s) that were not buggy** (Neto et al. 2018): Due to how SZZ works, it identifies all modified lines in a *BFC*. Some of these lines many not be related to the bug. Changes under this case do not modify the logic of the source code. For instance, the previous commit[30] of bug #4417 merged two different lines of code into one. But the logic of the code is still the same.
- **The previous commit made a reversion**: For instance, the previous commit[31] of bug #3274 was reverting a previous change. Thus, the commit that reverted the change cannot be the *BIC*.
- **The previous commit made a cosmetic change in a line that was not buggy** (da Costa et al. 2017; Neto et al. 2018; Kim et al. 2006): Due to how SZZ works, it can identify lines that were not buggy in previous modifications. Changes under this case include small cosmetic changes such as variable renaming or adding blank spaces to follow a coding style guide. For instance, one of the previous commits[32] of bug #8526 added a blank space between the equality sign and the value assigned to a variable. This previous commit did not introduce the bug; it is just a cosmetic change to refactor the source code.

Is there an Alternative Approach to find the *BIC* with Higher Accuracy? Previous approaches (Śliwerski et al. 2005; Kim et al. 2006; Williams and Spacco 2008) rely on the analysis of lines of code and assume that "a given bug is introduced by the lines of code that are modified to fix it". Thus, to determine the last revision that modified the lines fixed in a bug-fixing commit, researchers use features of the SCM systems such as "blame". Tools like blame only show the last change that modified the lines of code, but the source code lines may be modified several times. Thus, the disadvantage of using blame is that when a descendant change of a source code line introduced the bug, this change can be masked with posterior changes in the same line of the source code. In fact, according to Soetens et al., almost 25% of refactoring operations applied are masked when studying the version history of a software project at the commit level (Soetens et al. 2015).

Hence, an approach that increases the granularity of tools like blame may find *BIC*s with higher accuracy than the previous approaches (e.g., the four existing SZZ algorithms that we have studied previously). This alternative approach would track additions and deletions of tokens instead of additions and deletions of lines, so for every single token in the source code, this approach identifies the change that has last added/modified that token. Figure 11 shows a *BFC* analyzed using the line-based approach and Fig. 12 shows the same *BFC* analyzed using the token-based approach.

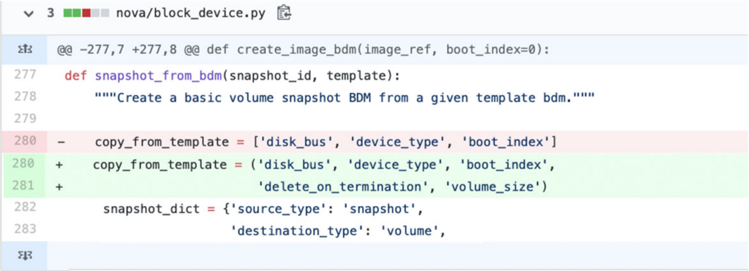
We will refer to the token-approach as *TSZZ* since it can be seen as a token-based SZZ approach. To evaluate whether the *TSZZ* approach increases the precision and recall when identifying *BIC*s, we analyzed the tokens that were modified in the *BFC* rather than the lines of the source code modified.

[29] https://github.com/elastic/elasticsearch/commit/9e4a0cba

[30] https://github.com/elastic/elasticsearch/commit/2e64dbce

[31] https://github.com/elastic/elasticsearch/commit/4c493ac

[32] https://github.com/elastic/elasticsearch/commit/5aa0a8438f



```
3 nova/block_device.py
@@ -277,7 +277,8 @@ def create_image_bdm(image_ref, boot_index=0):
  def snapshot_from_bdm(snapshot_id, template):
      """Create a basic volume snapshot BDM from a given template bdm."""

-     copy_from_template = ['disk_bus', 'device_type', 'boot_index']
+     copy_from_template = ('disk_bus', 'device_type', 'boot_index',
+                           'delete_on_termination', 'volume_size')
      snapshot_dict = {'source_type': 'snapshot',
                       'destination_type': 'volume',
```

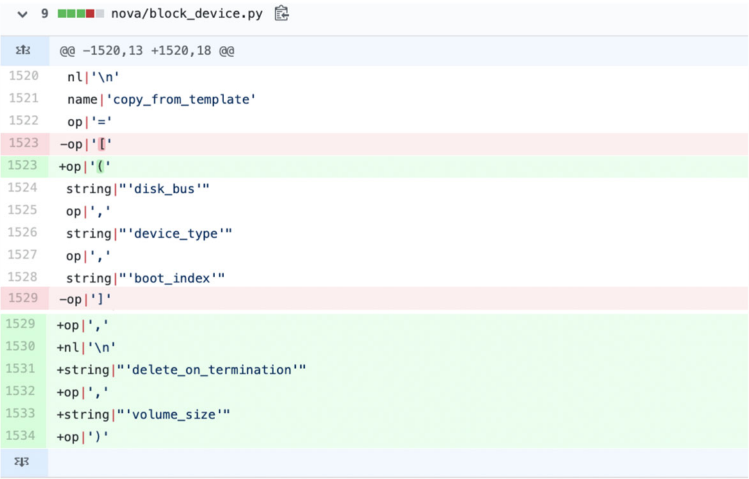
Fig. 11 BFC line based #1370177 of Nova

When we applied the TSZZ to the 46 *BFC*s of Nova it returned a set of 87 possible *BIC*s. When we applied TSZZ to the 43 *BFC*s of ElasticSearch it returned a set of 107 possible *BIC*s. Table 9 shows the values of precision, recall, and F-Score of the token-based algorithm. The table does not show the token-based counterpart of SZZ-1 because SZZ-1 uses annotation graphs (a line-based algorithm) and the result is the same as TSZZ.

The token-based SZZ solution slightly increases the precision and recall in Nova. However, in ElasticSearch this method performs worse, increasing the number of FN and FP, which decreases precision and recall.

8 Discussion

In this section we discuss the implications of our findings. First, we discuss to what extent our findings help towards establishing a bug introduction theory in the context of identifying the origin of bugs in open source projects (Section 8.1). Then, we discuss the generalizability of our findings (Section 8.3) and the implications with regard to the real evaluation of currents algorithms used during the bug identification process (Section 8.4). Finally, we discuss the threats to validity of this paper (Section 8.5).



```
9 nova/block_device.py
@@ -1520,13 +1520,18 @@
 nl|'\n'
 name|'copy_from_template'
 op|'='
-op|'['
+op|'('
 string|"'disk_bus'"
 op|','
 string|"'device_type'"
 op|','
 string|"'boot_index'"
-op|']'
+op|','
+nl|'\n'
+string|"'delete_on_termination'"
+op|','
+string|"'volume_size'"
+op|')'
```

Fig. 12 BFC token based #1370177 of Nova

Table 9 Results of True Positives *TP*, False Positives *FP*, False Negatives *FN*, Recall and Precision for the TSZZ-based, TSZZE-based and TSZZL-based algorithms

	Nova			ElasticSearch		
	TSZZ	TSZZE	TSZZL	TSZZ	TSZZE	TSZZL
TP	26 (27%)	21 (43%)	20 (40%)	24 (20%)	12 (21%)	16 (31%)
FP	61 (64%)	15 (30%)	16 (32%)	83 (68%)	18 (32%)	14 (27%)
FN	8 (9%)	13 (27%)	14 (28%)	14 (12%)	26 (47%)	22 (42%)
Precision	0.30	0.58	0.56	0.22	0.40	0.53
Recall	0.76	0.62	0.59	0.63	0.32	0.42
F-score	0.43	0.60	0.57	0.33	0.36	0.47

8.1 Towards a Better Understanding of Bug Introduction

The complex phenomenon of bug introduction has been studied before. Previous studies have helped researchers to understand that fixing bugs consist of determining why software behaves erroneously (Zeller 2009; Beller et al. 2018), that bugs can have different root causes (Li et al. 2006; Catolino et al. 2019), and that bugs can be introduced in a version of the software system but were not found until much later (Chen et al. 2014). However, the state-of-the-art lacks a better understanding of the origin of bugs. We believe that there are not enough empirical studies that attempt to define or evaluate how researchers can ensure that a change in the source code introduced a bug, the moment it was introduced.

Hence, researches assume that the lines of code that have been used to fix the bug were also the ones that introduced the bug in the first place is an inaccurate assumption that has been used in many studies. Furthermore, these studies implicitly assume that bugs have always been introduced by a developer. However, some recent studies (da Costa et al. 2017; Rodríguez-Pérez et al. 2018a, b) showed that this assumption should be reconsidered because other factors exist.

In our work, we have put this assumption aside and provided a model for ensuring when the software exhibits the bug and which change introduced it, in case that change exists. One of the most relevant contributions of the model is that it distinguishes between two different kind of bugs: intrinsic and extrinsic. The model relates intrinsic bugs with *BIC*s and extrinsic bugs with a fingerprint that the *BIC* does not exist.

Our model enables to understand the different ways in which bugs can be introduced. Practitioners can use it to describe the first time that the software exhibits the bug according to the *BFC*. Although our model is descriptive and defines many concepts and relationships, it cannot be understood as a theory of bug introduction because the lack of explicit prediction disqualifies it as a theory (Easterbrook et al. 2008; Gregor 2006). However, this work can be the starting point towards a better understanding of bug introduction because it goes beyond the mere observation of this phenomenon and tries to understand how and why this phenomenon occurs.

8.2 Guidelines for the Perfect Test Approximations Design

The perfect test provides a precise definition of “faulty code at the time of writing it”. This definition encompasses all the knowledge about the past software behavior, thus forming an oracle for each previous version; it also helps to describe whether a certain snapshot contains the bug fixed in the bug-fixing change. Although, this perfect test may be challenging to create because it is a theoretical construct, we can use some approximations to design it.

This section provides a guidelines to design these approximations based on our experience after our manual analysis of 116 bug reports. During this analysis we learned some lessons that would help assist researchers when designing perfect test approximations.

The Context Approximation This is the main source of information to design approximations for the perfect test. Descriptions and comments of a bug report provide a valuable knowledge about the context of the bug (e.g., bug cause, bug fix, bug symptoms ...), which helps researchers to decide whether there is a *BIC* or a *FFC*. Thus, when we understand the context of the bug, we can design the "perfect test" and analyze whether it would pass or fail in previous snapshots to find the *BIC* or the *FFC*. For example, the description of the bug report #2991[33] from ElasticSearch says:

> The BytesRefOrdValComparator uses Ordinals.Docs.getNumOrdinals() -1 as the upper bound for the binary search. The -1 causes that we ignore the last value in the segment.

and the description of the fix of this bug report says:

> Use full ord range in binary search. The upper bound of the binary search in BytesRefOrdValComparator starts at 1 and ends at maxOrd - 1. Yet, numOrd is defined as maxOrd - 1 excluding the 0 ord. This causes wrong sort ords when the bottom of the queue is compared to the next segment and the greatest term in the new segment is in-fact less than the current queue bottom.

With this information, we can mentally design an approximation for the "perfect test". It will test which snapshot, starting from the *BFC* backward, would fail because the source code of that snapshot causes wrong sort ords. Although, we cannot run this approximation automatically, to identify the *BIC*, we can manually analyze the source code of the previous snapshots and identify the first time that the test would fail. In some snapshots, we would not run the test because the function or feature tested is not present in that moment. In these cases, the first snapshot that fails after the test cannot run would be the *BIC* because the code was buggy when this snapshot introduced the function or feature tested.

The Modified Files Approximation When we do not have enough information to fully comprehend the context of the bug, we can also analyze the files modified by the *BFC* to understand whether the bug was caused by a *BIC* or a *FFC*. Either the name and the modified lines of some files can give us a useful hint to design the approximation for the "perfect test". For example, the bug fixing commit[34] from Nova modified the files: `doc/api-samples/versions-get-resp.json` and `nova/api/openstack/compute/views/versions.py`. Furthermore, the description of this *BFC* says:

> Apply v2.1 API to href of version API. Now Nova contains v2 and v2.1 APIs, but version API returns the same href between v2 and v2.1.

[33] https://github.com/elastic/elasticsearch/issues/2991

[34] https://opendev.org/openstack/nova/commit/46bd4e4292648c0474e02ddc1560ce583fbe56d0

With this information, we can mentally design an approximation for the “perfect test”. This approximation will test which snapshot, starting from the *BFC* backward, would fail because the source code of that snapshot returns a wrong API version. In this case, we will notice that, based on the definition of “the perfect test”, there is no faulty code at the time of writing it. Thus, the test would always pass, which indicates that there is no *BIC* but a *FFC* because the bug was caused by the evolution of the code. After adding a new version of the API, the bug manifested itself in the source code causing the URL links to not show correctly. Sometimes, when we have enough information, we manually can point out which is the *FFC*. However, in most of the cases, we cannot identify the *FFC* because the developers do not give such information; and it is difficult to manually identify this change navigating from the *BFC* backward.

The Bug Live Period Approximation In addition to the context, in some cases, we can analyze some metadata such as the date of the snapshots. With this information we can compute how long the bug has survived in the source code until it was reported in the issue tracking system, previous studies suggested that this period should not be bigger than two years (da Costa et al. 2017; Rodriguez-Perez et al. 2017; Chen et al. 2014). For example, when we analyze the previous snapshots to identify whether the “perfect test” would pass or not, we can also analyze the time period between the bug report date and the date of the snapshot. If this period spans more than two years, we can assume that the source code at the time of writing it was correct, thereby, the “perfect test” would pass.

8.3 Generalizability of our Findings

The process of operationalizing the model in two different projects leads us to obtain a method to identify the first time that the software fails according to a *BFC*. We think that the case studies selected in this article are so different that this method can be generalized. Thus, researchers can apply this method in other projects in order to build reliable datasets that contain the information about the *BIC*s.

By using ElasticSearch and Nova as case studies, we gain deep insights into how bugs manifest themselves for the first time in these projects. They are exploratory case studies as we do not have a theory to refute or circumspect. However, the empirical results in Section 7.2 demonstrate that the current assumption –“a bug was introduced by the lines of code that were modified to fix it”– is just one of the cases among others of how bugs were introduced in software.

First, 21% of the bugs analyzed in Nova and 9% in ElasticSearch are extrinsic, meaning that they do not have a change that introduced the bug directly in the SCM. We hypothesize that the reason why the percentage of extrinsic bugs is higher in Nova is due to the nature of the software and its changing environments. It should be noted that Nova, in contrast to ElasticSearch, is infrastructure software, that runs at the OS level and on many different platforms, which leads us to think that situations that end in extrinsic bugs appear more frequently. However, we do not have evidence to demonstrate what specific characteristics of software can contribute more to this difference.

Second, in both projects, the F-score of the four existing SZZ algorithms aimed at determining the origin of bugs varies from 0.44 to 0.77 depending on the criteria that we use to evaluate the SZZ-based algorithms. The assumption “a bug was introduced by the lines of code that were modified to fix it” is one of many cases when a bug is introduced; in our manual analysis, we found that this holds true for only about 61% of the cases in the best scenario. The bugs that were not introduced by the lines of code that were modified to fix them were identified as false positives, some of the reasons of being false positives were refactoring changes, reverting commits or equivalent changes, among others.

Hence, it is comprehensible to think that these results can be generalized to other projects. Thus, if we analyze how bugs were introduced in other projects we will find that a percentage of them are being caused by factors different from a developer introducing buggy code in the software.

8.4 Drawbacks of Existing Algorithms and Benefits of the Proposed Model to Software Engineering

Over the past decades, researchers have used datasets obtained from SZZ-based algorithms to feed their bug prediction or classification models. For example, Ray et al. used a dataset gathered using the SZZ algorithm (Rahman et al. 2014) to study the naturalness of buggy code (Ray et al. 2016). Massacci et al. evaluated most existing vulnerabilities discovery models on web browsers and took many datasets obtained using SZZ (Massacci and Nguyen 2014). Abreu et al. used the dataset obtained in Śliwerski et al. (2005) to study how the frequency of communication between developers affects the introduction of a bug in the source code (Abreu and Premraj 2009). These datasets can contain a noteworthy number of false positives and false negatives as we have seen in the findings of our case studies (see Section 7.2.2). Consequently, the results of previous studies in the larger domain of software engineering (e.g., bug prediction or bug detection) can differ (negatively) if we take into account that they have used those datasets.

This work demonstrates that the process of applying our model to 116 bug reports and analyzing 236 previous commits leads to reliable datasets in which each *BFC* is linked with its *BIC* or without one. These curated datasets are one of the benefits of using the model as they represent the ground truth of the projects and they could be crucial to improve other areas of software engineering.

In this work, we manually built these curated datasets, and then we used them to compute the real performance in terms of precision, recall, and F-score of four SZZ-based algorithms. The results show that: (i) there are intrinsic and extrinsic bugs, although the SZZ-based algorithms consider all bugs as the same; (ii) the correct identification of *BIC*s is still a challenge when using SZZ-based algorithms; (iii) specific characteristics of the project might affect the performance of the algorithms when identifying *BIC*s. For example, we have noticed that the SZZ-1E algorithm obtains the best performance in Nova, while the SZZ-1L algorithm did in ElasticSearch; (iv) the existence of extrinsic bugs is a crucial factor for the performance of these algorithms: when they are removed from the dataset, the performance of these algorithms increases. We have also shown that researchers can decide what criteria they prefer to use when evaluating the SZZ algorithms depending on different factors. For example, if they attempt to analyze which algorithm creates a better dataset of false positives, they can decide to use the "All" criteria. Also, they can use the "At least" criteria to analyze which algorithm identifies more *BIC*s. Finally, they may prefer to use the "Only" criteria to evaluate whether just one change introduced the bug.

After the manual analysis, we have realized that establishing whether a *BIC* exists, and determining when it was introduced is not straightforward. However, the proposed model helps to identify the first time that the software exhibits the bug and to understand whether it was a *BIC* or a *FFC*. This model not only provides guidelines on how to become operational in real projects to build reliable datasets, it also contemplates *BFC*s that have been largely not considered in the current research literature. For example, the *BFC*s with only new lines added are not considered in the current research literature because the SZZ-based algorithm cannot track back these lines. Thus, another benefit of this paper is that our model decreases the number of false negatives in the datasets because it considers all kinds of *BFC*s.

All in all, we believe that the proposed model greatly benefits software engineering, as for the first time, we have described when a software system exhibits a bug, and we have looked into how bugs were inserted. In addition, with the empirical evaluation of the proposed model and the evaluation of the effectiveness of SZZ-based algorithms, we shed some more light on the problem of identifying software bugs realistically. However, to achieve greater bug localization automation, we need a concerted effort in testing to find ways or techniques to address the challenges of making the model operational (see Section 6). In particular, a (partially) automated technique for building and subsequently evolving a *perfect test* would be of great importance, as it is this test that can signal the bug and then find the *BIC* or the *FFC*.

8.5 Threats to Validity

The validity of our work is described in terms of the four main threats to validity in empirical software engineering research: construct, internal, external and conclusion validity (Wohlin et al. 2012).

Construct Validity Since we do not have enough means to build or automatize the perfect test, we have to create it mentally and this can lead to some threats in the results. However, we mitigate this threat by discussing those cases in which we were unsure about how the perfect test should be implemented. However, if a bug exists and it is fixed, then a test can be created to show the existence/lack of a bug. Otherwise, researchers cannot know for sure if the bug was fixed.

Also, assuming that the bug reports analyzed were not reopened later and that their *BFC* was always complete or that there is no duplicates of the same bug may be a threat to the study. We try to mitigate these cases by analyzing whether the *BFC*s have one or more *BFC*s attached to them or whether there was any information in the bug tracker system about the bug reports being reopened. Also, there can be cases where commits detected as *BFC*s turn out to be false because the bug report did not describe a real bug. To reduce this threat, the *BFC*s were manually reviewed to filter out the uncertain cases. Also, we manually located the *BIC* and in order to compute the performance of SZZ-based algorithms, we removed those bugs for which we were unsure from the datasets. Other threats are related to the peculiarities of the projects. The use of *GNU Diffutils* is the most extended way of providing diff information when looking for the difference between two files. However, other ways of providing diff information can be considered.

Internal Validity The most important internal threat is that the authors, although they know OpenStack and ElasticSearch from using and having previously investigated them, do not have advanced programming expertise in these systems. This may have influenced the results of the analysis. To mitigate, the cases where we were unsure were discussed among the authors of this paper and removed when no agreement was reached.

External Validity In terms of the number of commits that we analyzed for our study, it should be noted that our numbers are in the order of magnitude of similar studies that require intensive human labor, Hindle et al. considered 100 large commits in their study (Hindle et al. 2008), Da Costa et al. analyzed 160 bugs and 80 *BIC*s (2017), and Williams and

Spacco studied 25 *BFC*s that contained a total of 50 changed lines which were mapped back to a *BIC* (2008).

Another threat is that this work has only selected two different programming languages, Java and Python. It is possible that the study of different programming languages leads to different results. The use of Nova and ElasticSearch as the case studies implies a better understanding of how bugs appear in these projects. However, a higher number of projects would enrich the study because Nova and ElasticSearch can have specific properties. Both have rapid evolution and an active community of developers, thus other projects with fewer commits per year could have different results.

Conclusion Validity The metrics used to evaluate the four existing SZZ algorithms (i.e., accuracy, precision, recall, and F1-Score) are widely used when evaluating the performance of algorithms that identify the origin of bugs (Davies et al. 2014). Not having used or compared all the existing SZZ-based approaches (e.g., RA-SZZ (Neto et al. 2018) can be a threat to the conclusion validity of the study since these approaches may have better accuracy and precision. Although comparing our manually curated dataset with other SZZ-based approaches would give us more insights into the performance of those approaches, we discarded using them because of the complexity of implementing them and the unavailability to use them as open source software. However, we study the token-based approach because we believe that it would have better precision that the four SZZ-based approaches that we selected.

9 Conclusions and Future Work

In this study, to answer our central question: *How can we identify the origin of a defect based on information in source control systems?*, we proposed a model for defining criteria to decide the first snapshot of an evolving software system that exhibits a certain bug. For that, the model defines “the perfect test”, which fails when the bug is observed after a change to the software and passes when the bug in not observed. In practice, this “perfect test” can be (mentally) created using information from the source control systems, issue tracker systems and code review systems.

When applying the criteria to two real world projects, we qualitatively show that in the 116 bugs that we consider it is not always straightforward to identify how bugs were introduced. Furthermore, we witnessed how some bugs were caused by changes or omissions in the source code of the project (60%–64%). Other bugs (i.e., the extrinsic ones) were caused by changes that are not recorded in the source code (9%–21%). The proposed model helps to distinguish both cases and identifies when the *BIC* was made. The evaluation of four existing SZZ algorithms shows that when a change in the source code caused the bug, the assumption “a bug was introduced by the lines of code that were modified to fix it” only holds for 61%-63% of the commits analyzed, in the best case of SZZ-1. The precision does not exceed 0.75 and the maximum value for the recall is 0.80 in the projects that we evaluated. Furthermore, the results show that the version of SZZ with a higher effectiveness is SZZ-1 when using the “At least” criteria.

The lion’s share of identifying the bug-introducing changes is based on techniques which rely on the assumption that the lines of code changed to fix the bug are also the ones that have introduced it. This work provides evidence of the problematic nature of this assumption, and demonstrates that it is just one of the cases among others of how bugs were introduced in software components. Potentially, this finding has many implications in other fields of software engineering (e.g., bug prediction or bug detection) since many studies are misidentifying or even omitting the origin of the bug and this can put their results in jeopardy. This work does not try to make a formal proposal of a theory that explains how bugs were introduced in software products since we cannot be predictive. However, considering the apparent suitability of the model proposed, and the implications of the findings, it seems it could be necessary to obtain such a theory and, this work serves as a motivation towards a theory bug introduction. Our work also contributes to this by defining and explaining all relevant concepts in bug introduction, proposing a unified terminology.

We have demonstrated that our model enables to identify the snapshot of a component that exhibits the bug. Future work could use this model to build more datasets that can be used as the *ground truth* to evaluate the real performance of techniques when identifying how bugs were introduced. In order to build these datasets faster, another interesting and useful line of research would be to automate the *perfect test* that signals whether the bug is present in the code.

The findings in this study show that there are two kind of bugs, intrinsic bugs (the origin is a *BIC* in the SCM), and extrinsic bugs (the origin is a change not recorded in the SCM). Furthermore, the findings show that four existing SZZ algorithms misidentify *BIC*s. Other future lines could be i) to study whether extrinsic bugs can be automatically detected, and ii) to assess the impact of misidentifying *BIC*s in other areas of software engineering such as automatic bug detection or bug prediction. This could help to better design integration tests, or to envision other procedures to make software development more robust against bugs.

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

Replication Package we have set up a replication package[35] including data sources, intermediate data and scripts.

Acknowledgments We want to express our gratitude to Bitergia[36] for the support they have provided when questions have arisen using their tools. We also acknowledge the support of several authors by the Government of Spain through projects TIN2014-59400-R and “BugBirth” RTI2018-101963-B-I00. The first author has been supported by the 4TU federation (The Netherlands) through the project “Social aspects of software quality”. Other funding came from the Netherlands Organisation for Scientific Research (NWO) through the “TestRoots” project and the EU Horizon 2020 ICT-10-2016-RIA “STAMP” project (No.731529).

References

Abreu R, Premraj R (2009) How developer communication frequency relates to bug introducing changes. In: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops. ACM, pp 153–158.

[35] https://github.com/Gemarodri/HowBugsAreBorn

[36] http://bitergia.com

Alexandru CV, Gall H (2015) Rapid multi-purpose, multi-commit code analysis. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), vol 2. IEEE, pp 635–638

Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: Proceedings of the 31st international conference on software engineering. IEEE Computer Society, pp 298–308

Basili VR, Perricone BT (1984) Software errors and complexity: an empirical investigation. Commun ACM 27(1):42–52

Bavota G, Russo B. (2015) Four eyes are better than two: on the impact of code reviews on software quality. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp 81–90

Beller M, Gousios G, Zaidman A (2015) How (much) do developers test? In: Proceedings of the International Conference on Software Engineering (ICSE), vol 2. IEEE Computer Society, pp 559–562

Beller M, Gousios G, Zaidman A (2017) Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In: Proceedings of the International Conference on Mining Software Repositories (MSR). IEEE, pp 356–367

Beller M, Spruit N, Spinellis D, Zaidman A (2018) On the dichotomy of debugging behavior among programmers. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp 572–583

Berger ED, Hollenbeck C, Maj P, Vitek O, Vitek J (2019) On the impact of programming languages on code quality. arXiv:1901.10220

Bettenburg N, Hassan AE (2013) Studying the impact of social interactions on software quality. Empir Softw Eng 18(2):375–431

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, pp 121–130

Bird C, Rigby PC, Barr E, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: 6th IEEE international working conference on mining software repositories, 2009. MSR'09. IEEE, pp 1–10

Bissyande TF, Thung F, Wang S, Lo D, Jiang L, Reveillere L (2013) Empirical evaluation of bug linking. In: 2013 17th european conference on software maintenance and reengineering (CSMR). IEEE, pp 89–98

Boehm B, Basili VR (2005) Software defect reduction top 10 list. Foundations of empirical software engineering: the legacy of Victor R Basili 426(37):426–431

Brooks FP Jr (1995) The mythical man-month: essays on software engineering, anniversary edition, 2/E. Pearson Education India

Canfora G, Cerulo L, Cimitile M, Di Penta M (2011) Social interactions around cross-system bug fixings: the case of freebsd and openbsd. In: Proceedings of the 8th working conference on mining software repositories. ACM, pp 143–152

Catolino G, Palomba F, Zaidman A, Ferrucci F (2019) Not all bugs are the same Understanding, characterizing, and classifying bug types. J Syst Softw 152:165–181

Chandy KM, Lamport L (1985) Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS) 3(1):63–75

Chen T.-H., Nagappan M, Shihab E, Hassan AE (2014) An empirical study of dormant bugs. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 82–91

Čubranic D, Murphy GC (2003) Hipikat: Recommending pertinent software development artifacts. In: 25th international conference on software engineering, 2003. Proceedings. IEEE, pp 408–418

da Costa DA, Kulesza U, Aranha E, Coelho R (2014) Unveiling developers contributions behind code commits: an exploratory study. In: Proceedings of the 29th annual ACM symposium on applied computing. ACM, pp 1152–1157

da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2017) A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans Softw Eng 43(7):641–657

Davies S, Roper M, Wood M (2014) Comparing text-based and dependence-based approaches for determining the origins of bugs. Journal of Software: Evolution and Process 26(1):107–139

Demeyer S, Ducasse S, Nierstrasz O (2002) Object-oriented reengineering patterns. Elsevier

Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S (2015) Untangling fine-grained code changes. In: 2015 IEEE 22nd international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 341–350

Easterbrook S, Singer J, Storey M.-A., Damian D (2008) Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering. Springer, pp 285–311

Ebert F, Castor F, Serebrenik A (2015) An exploratory study on exception handling bugs in Java programs. J Syst Softw 106:82–101

Ell J (2013) Identifying failure inducing developer pairs within developer networks. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 1471–1473

Fischer M, Pinzger M, Gall H (2003) Analyzing and relating bug report data for feature tracking. In: Proceedings of the 10th working conference on reverse engineering (WCRE). IEEE, pp 90–100

Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: International conference on software maintenance, 2003. ICSM 2003. Proceedings. IEEE, pp 23–32

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional

Fraser G, Arcuri A (2013a) Evosuite: On the challenges of test case generation in the real world. In: 2013 IEEE sixth international conference on software testing, verification and validation (ICST). IEEE, pp 362–369

Fraser G, Arcuri A (2013b) Whole test suite generation. IEEE Trans Softw Eng 39(2):276–291

German DM, Hassan AE, Robles G (2009) Change impact graphs: Determining the impact of prior codechanges. Inf Softw Technol 51(10):1394–1408

Gonzalez-Barahona JM, Robles G, Herraiz I, Ortega F (2014) Studying the laws of software evolution in a long-lived floss project. J Softw Evol Process 26(7):589–612

Gousios G, Zaidman A, Storey MD, van Deursen A (2015) Work practices and challenges in pull-based development: the integrator's perspective. In: Bertolino A., Canfora G., Elbaum S. G. (eds) 37th IEEE/ACM international conference on software engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, vol 1. IEEE Computer Society, pp 358–368

Gregor S (2006) The nature of theory in information systems. MIS quarterly:611–642

Guerrouj L, Kermansaravi Z, Arnaoudova V, Fung BC, Khomh F, Antoniol G, Guéhéneuc Y-G (2015) Investigating the relation between lexical smells and change-and fault-proneness: an empirical study. Softw Qual J:1–30

Hassan AE (2009) Predicting faults using the complexity of code changes. In: IEEE 31st international conference on software engineering, 2009. ICSE 2009. IEEE, pp 78–88

Hassan AE, Holt R (2005) The top ten list: Dynamic fault prediction. In: Proceedings of the 21st IEEE international conference on software maintenance, 2005. ICSM'05. IEEE, pp 263–272

Hata H, Mizuno O, Kikuno T (2010) Fault-prone module detection using large-scale text features based on spam filtering. Empir Softw Eng 15(2):147–165

Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 392–401

Hilton M, Bell J, Marinov D (2018) A large-scale study of test coverage evolution. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering (ASE). ACM, pp 53–63

Hindle A, German DM, Holt R (2008) What do large commits tell us?: a taxonomical study of large commits. In: Proceedings of the 2008 international working conference on mining software repositories. ACM, pp 99–108

Institute of Electrical and Electronics Engineers and IEEE Computer Society. Software Engineering Standards Committee (2009) IEEE standard 1044-2009: Classification for software anomalies. IEEE std IEEE

ISO/IEC (2001) ISO/IEC 9126. Software engineering – Product quality ISO/IEC

Itkonen J, Mantyla MV, Lassenius C (2007) Defect detection efficiency: Test case based vs. exploratory testing. In: First international symposium on empirical software engineering and measurement, 2007. ESEM 2007. IEEE, pp 61–70

Izquierdo-Cortazar D, Capiluppi A, Gonzalez-Barahona JM (2011) 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

Izquierdo-Cortázar D., Robles G, González-Barahona J. M. (2012) Do more experienced developers introduce fewer bugs? In: IFIP International conference on open source systems. Springer, pp 268–273

Jacobs J, Van Moll J, Kusters R, Trienekens J, Brombacher A (2007) Identification of factors that influence defect injection and detection in development of software intensive products. Inf Softw Technol 49(7):774–789

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 92–101

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773

Kim S, Whitehead E Jr et al (2006) Properties of signature change patterns. In: 22nd IEEE international conference on software maintenance, 2006. ICSM'06. IEEE, pp 4–13

Kim S, Whitehead E Jr, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Trans Softw Eng 34(2):181–196

Kim S, Zimmermann T, Pan K, Whitehead E Jr (2006) Automatic identification of bug-introducing changes. In: 21st IEEE/ACM international conference on automated software engineering, 2006. ASE'06. IEEE, pp 81–90

Kim S, Zimmermann T, Whitehead E Jr, Zeller A (2007) Predicting faults from cached history. In: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, pp 489–498

Knauss E, Damian D, Knauss A, Borici A (2014) Openness and requirements: opportunities and tradeoffs in software ecosystems. In: 2014 IEEE 22nd international requirements engineering conference (RE). IEEE, pp 213–222

Knauss E, Liebel G, Horkoff J, Wohlrab R, Kasauli R, Lange F, Gildert P (2018) T-reqs: Tool support for managing requirements in large-scale agile system development. arXiv:1805.02769

LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th international conference on Software engineering. ACM, pp 492–501

Le T-DB, Linares-Vásquez M, Lo D, Poshyvanyk D (2015) Rclinker: automated linking of issue reports and commits leveraging rich contextual information. In: 2015 IEEE 23rd international conference on program comprehension (ICPC). IEEE, pp 36–47

Li Z, Tan L, Wang X, Lu S, Zhou Y, Zhai C (2006) Have things changed now?: an empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability. ACM, pp 25–33

Marsavina C, Romano D, Zaidman A (2014) Studying fine-grained co-evolution patterns of production and test code. In: 2014 IEEE 14th international working conference on source code analysis and manipulation (SCAM). IEEE, pp 195–204

Massacci F, Nguyen VH (2014) An empirical methodology to evaluate vulnerability discovery models. IEEE Trans Softw Eng 40(12):1147–1162

Meneely A, Srinivasan H, Musa A, Tejeda AR, Mokary M, Spates B (2013) When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In: 2013 ACM/IEEE international symposium on empirical software engineering and measurement. IEEE, pp 65–74

Mens T, Wermelinger M, Ducasse S, Demeyer S, Hirschfeld R, Jazayeri M (2005) Challenges in software evolution. In: Eighth international workshop on principles of software evolution. IEEE, pp 13–22

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Technical Journal 5(2):169–180

Moonen L, van Deursen A, Zaidman A, Bruntink M (2008) On the interplay between software testing and evolution and its effect on program comprehension. In: Mens T, Demeyer S (eds) Software evolution. Springer, pp 173–202

Murphy-Hill E, Zimmermann T, Bird C, Nagappan N (2015) The design space of bug fixes and how developers navigate it. IEEE Trans Softw Eng 41(1):65–81

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th international conference on Software engineering. ACM, pp 452–461

Nakajo T, Kume H (1991) A case history analysis of software error cause-effect relationships. IEEE Trans Softw Eng 17(8):830–838

Neto EC, da Costa DA, Kulesza U (2018) The impact of refactoring changes on the SZZ algorithm: an empirical study. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 380–390

Neto EC, da Costa DA, Kulesza U (2019) Revisiting and improving SZZ implementations. In: 2019 ACM/IEEE International symposium on empirical software engineering and measurement. ACM

Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Proceedings of the conference on the future of software engineering. ACM, pp 35–46

Ottenstein KJ, Ottenstein LM (1984) The program dependence graph in a software development environment. ACM Sigplan Notices 19(5):177–184

Palomba F, Panichella A, Zaidman A, Oliveto R, De Lucia A (2016) Automatic test case generation: what if test code quality matters? In: Proceedings of the 25th international symposium on software testing and analysis (ISSTA). ACM, pp 130–141

Palomba F, Zaidman A (2017) Does refactoring of test smells induce fixing flaky tests? In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp 1–12

Palomba F, Zaidman A (2019) The smell of fear: on the relation between test smells and flaky tests. Empirical Software Engineering (EMSE) 24(5):2907–2946

Pan K, Kim S, Whitehead EJ Jr (2009) Toward an understanding of bug fix patterns. Empir Softw Eng 14(3):286–315

Prechelt L, Pepper A (2014) Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. Inf Softw Technol 56(10):1377–1389

Rahman F, Khatri S, Barr E, Devanbu P (2014) Comparing static bug finders and statistical prediction. In: Proceedings of the 36th international conference on software engineering. ACM, pp 424–434

Rahman F, Posnett D, Hindle A, Barr E, Devanbu P (2011) Bugcache for inspections: hit or miss? In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering. ACM, pp 322–331

Ray B, Hellendoorn V, Godhane S, Tu Z, Bacchelli A, Devanbu P (2016) On the naturalness of buggy code. In: Proceedings of the 38th international conference on software engineering. ACM, pp 428–439

Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in github. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 155–165

Rodríguez-Pérez G., Gonzalez-Barahona JM, Robles G, Dalipaj D, Sekitoleko N (2016) Bugtracking: A tool to assist in the identification of bug reports. In: IFIP International conference on open source systems. Springer, pp 192–198

Rodriguez-Perez G, Robles G, Gonzalez-Barahona JM (2017) How much time did it take to notify a bug?: two case studies: elasticsearch and nova. In: Proceedings of the 8th workshop on emerging trends in software metrics. IEEE Press, pp 29–35

Rodríguez-Pérez G, Robles G, González-Barahona JM (2018a) Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. Inf Softw Technol 99:164–176

Rodríguez-Pérez G, Zaidman A, Serebrenik A, Robles G, González-Barahona JM (2018b) What if a bug has a different origin? Making sense of bugs without an explicit bug introducing change. In: 12th international symposium on empirical software engineering and measurement (ESEM). ACM, pp 52:1–52:4

Rothermel G, Harrold MJ (1996) Analyzing regression test selection techniques. IEEE Trans Softw Eng 22(8):529–551

Runeson P, Host M, Rainer A, Regnell B (2012) Case study research in software engineering: Guidelines and examples. Wiley, New York

Sahal E, Tosun A (2018) Identifying bug-inducing changes for code additions. In: Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement. ACM, p 57

Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto K.-i. (2013) Studying re-opened bugs in open source software. Empir Softw Eng 18(5):1005–1042

Sinha VS, Sinha S, Rao S (2010) Buginnings: identifying the origins of a bug. In: Proceedings of the 3rd India software engineering conference. ACM, pp 3–12

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes?. Proceedings of the 2005 International workshop on mining software repositories: 1–5

Soetens QD, Pérez J., Demeyer S, Zaidman A (2015) Circumventing refactoring masking using fine-grained change recording. In: Proceedings of the 14th international workshop on principles of software evolution. ACM, pp 9–18

Sun Y, Wang Q, Yang Y (2017) Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance. Inf Softw Technol 84:33–47

Syer MD, Nagappan M, Adams B, Hassan AE (2015) Replicating and re-evaluating the theory of relative defect-proneness. IEEE Trans Softw Eng 41(2):176–197

Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empir Softw Eng 19(6):1665–1705

Tantithamthavorn C, Teekavanich R, Ihara A, Matsumoto K-i (2013) Mining a change history to quickly identify bug locations: a case study of the eclipse project. In: 2013 IEEE international symposium on software reliability engineering workshops (ISSREW). IEEE, pp 108–113

Thung F, Lo D, Jiang L (2013) Automatic recovery of root causes from bug-fixing changes. In: 2013 20th working conference on reverse engineering (WCRE). IEEE, pp 92–101

Vassallo C, Zampetti F, Romano D, Beller M, Panichella A, Penta MD, Zaidman A (2016) Continuous delivery practices in a large financial organization. In: Proceedings of the international conference on software maintenance and evolution (ICSME). IEEE Computer Society, pp 519–528

Viller S, Bowers J, Rodden T (1999) Human factors in requirements engineering: A survey of human sciences literature relevant to the improvement of dependable systems development processes. Interact Comput 11(6):665–698

Vonken F, Zaidman A (2012) Refactoring with unit testing: a match made in heaven?. In: Proceeedings of the working conference on reverse engineering (WCRE), pp 29–38

Weiss C, Premraj R, Zimmermann T, Zeller A (2007) How long will it take to fix this bug? In: Fourth international workshop on mining software repositories, 2007. ICSE Workshops MSR'07. IEEE, pp 1–1

Williams C, Spacco J (2008) SZZ revisited: Verifying when changes induce fixes. In: Proceedings of the 2008 workshop on defects in large software systems. ACM, pp 32–36

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer Science & Business Media

Wu R, Zhang H, Kim S, Cheung S-C (2011) 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. ACM, pp 15–25

Xia X, Lo D, Shihab E, Wang X, Zhou B (2015) Automatic, high accuracy prediction of reopened bugs. Autom Softw Eng 22(1):75–109

Zaidman A, Van Rompaey B, Demeyer S, van Deursen A (2008) Mining software repositories to study co-evolution of production & test code. In: First international conference on software testing, verification, and validation (ICST). IEEE, pp 220–229

Zaidman A, Van Rompaey B, van Deursen A, Demeyer S (2011) Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empir Softw Eng 16(3):325–364

Zeller A (2009) Why programs fail - a guide to systematic debugging, 2nd edn. Academic Press, New York

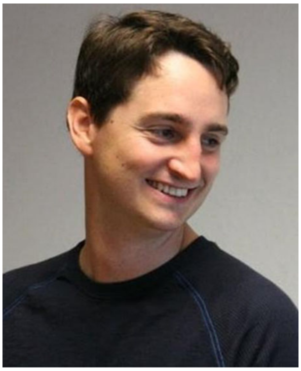
Zeller A, Hughes W, Lavery J, Doran K, Morrison CT, Snodgrass RT, Stärk RF (2011) Causes and effects in computer programs. In: Proceedings of the fifth international workshop on computer, pp 482–508

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for Eclipse. In: International workshop on predictor models in software engineering, 2007. PROMISE'07: ICSE workshops 2007. IEEE, pp 9–9

Zimmermann T, Nagappan N, Guo PJ, Murphy B (2012) Characterizing and predicting which bugs get reopened. In: Proceedings of the 34th international conference on software engineering. IEEE Press, pp 1074–1083

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Gema Rodríguez-Pérez is a Postdoctoral Researcher at the University of Waterloo, Canada. She received a European Ph.D. degree in Computer Science from the Universidad Rey Juan Carlos, Spain, in 2018. Her research interests include bug introducing changes, empirical software engineering, software maintenance and evolution, and mining software repositories. She is a member of the IEEE. More about Gema and her work is available online at https://gemarodri.github.io/PersonalWeb/.



Gregorio Robles is Associate Professor at the Universidad Rey Juan Carlos, a public university near Madrid, Spain. Gregorio is specialized in free/open source software research. He is one of the founders of Bitergia. a software development analytics company, and has been the General Chair of IEEE ICSME in 2018, and will be the General Chair of MSR in 2021.

Alexander Serebrenik is an associate professor of software evolution at the Eindhoven University of Technology, The Netherlands. His research interests include a wide range of software maintenance and evolution topics, from source-code analysis to collaborative and human aspects of software engineering. Serebrenik received a Ph.D. in computer science from Katholieke Universiteit Leuven, Belgium. He has coauthored “Evolving Software Systems” (Springer Verlag, 2014) and more than 170 scientific papers and articles. He is a Senior Member of the IEEE. Contact him at a.serebrenik@tue.nl.

Andy Zaidman is a full professor in software engineering at Delft University of Technology, the Netherlands. He obtained his MSc (2002) and PhD (2006) in Computer Science from the University of Antwerp, Belgium. His main research interests are software evolution, program comprehension, mining software repositories and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE’08, WCRE’09, VISSOFT’14 and MSR’18). He is on the editorial board of JSS and EMSE. In 2013 he was the laureate of a Vidi career grant from the Dutch science foundation NWO, while in 2019 he won the Vici career grant, the most prestigious career grant from the Dutch science foundation NWO.

Daniel M. Germán is Professor in the Department of Computer Science at the University of Victoria, where he does research in the areas of mining software repositories, open source software engineering, and intellectual property.

Jesus M. Gonzalez-Barahona teaches and researches in Universidad Rey Juan Carlos, Fuenlabrada (Spain). His research interests include the study of software development communities and processes, with a focus on quantitative analysis. He is one of the founders of Bitergia, the software development analytics company. He likes coding, taking photos and drinking coffee.

Affiliations

Gema Rodríguez-Pérez[1] · Gregorio Robles[2] · Alexander Serebrenik[3] · Andy Zaidman[4] · Daniel M. Germán[5] · Jesus M. Gonzalez-Barahona[2]

Gema Rodríguez-Pérez
gema.rodriguez-perez@uwaterloo.ca

Alexander Serebrenik
a.serebrenik@tue.nl

Andy Zaidman
a.e.zaidman@tudelft.nl

Daniel M. Germán
dmg@uvic.ca

Jesus M. Gonzalez-Barahona
jgb@gsyc.urjc.es

[1] University of Waterloo, Waterloo, ON N2L 3G1, Canada

[2] Universidad Rey Juan Carlos, Madrid, Spain

[3] Eindhoven University of Technology, Eindhoven, The Netherlands

[4] Delft University of Technology, Delft, The Netherlands

[5] University of Victoria, Victoria, BC V8P 5C2, Canada