Noname manuscript No.
(will be inserted by the editor)

An Empirical Study on the Use of SZZ for Identifying Inducing Changes of Non-functional Bugs

Sophia Quach · Maxime Lamothe · Yasutaka Kamei · Weiyi Shang

Received: date / Accepted: date

Abstract Non-functional bugs, e.g., performance bugs and security bugs, bear a heavy cost on both software developers and end-users. For example, IBM estimates the cost of a single data breach to be millions of dollars. Tools to reduce the occurrence, impact, and repair time of non-functional bugs can therefore provide key assistance for software developers racing to fix these issues. Identifying bug-inducing changes is a critical step in software quality assurance. In particular, the SZZ approach is commonly used to identify bug-inducing commits. However, the fixes to non-functional bugs may be scattered and separate from their bug-inducing locations in the source code. The nature of non-functional bugs may therefore make the SZZ approach a sub-optimal approach for identifying bug-inducing changes. Yet, prior studies that leverage or evaluate the SZZ approach do not consider non-functional bugs, leading to potential bias on the results.

In this paper, we conduct an empirical study on the results of the SZZ approach when used to identify the inducing changes of the non-functional bugs in the NFBugs dataset. We eliminate a majority of the bug-inducing commits as they are not in the same method or class level. We manually examine whether each identified bug-inducing change is indeed the correct bug-inducing change. Our manual study shows that a large portion of non-functional bugs cannot be properly identified by the SZZ approach. By manually identifying the root causes of the falsely detected bug-inducing changes, we uncover root causes for false detection that have not been found by previous studies. We evaluate the identified bug-inducing changes based on three criteria from prior research, i.e., the earliest bug appearance, the future impact of changes, and the realism of bug introduction. We find that prior criteria may be irrelevant for non-functional bugs. Our results may be used to assist in future research on non-functional bugs, and highlight the need

S.Quach · M.Lamothe · W.Shang
Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada
E-mail: {s_quach,max_lam,shang}@encs.concordia.ca

Y.Kamei
Faculty of Information Science and Electrical Engineering, Kyushu University, Japan
E-mail: kamei@ait.kyushu-u.ac.jp

to complement SZZ to accommodate the unique characteristics of non-functional bugs.

Keywords Bug inducing changes SZZ · non-functional bugs · mining software repositories

1 Introduction

Since the first software bug was discovered in 1945, software engineers have been developing techniques to attempt to fix and prevent them [1–3]. Bugs are costly to fix, and increase maintenance effort [4]. In a world that is ever more reliant on software [5], it appears more important than ever before to have quality software and to be able to fix bugs in a timely manner when they do appear. To this end, researchers have developed several approaches to identify prior bug-inducing changes to help development teams avoid future bugs by learning from their mistakes [1–3].

One technique to help with bug localization, called the SZZ approach, attempts to find the source-code changes that first induces a software bug [6,7]. However, the SZZ approach, like other bug localization techniques [8], is not perfect. Previous studies have shown that the SZZ approach can mislabel some changes as bug-inducing [6]. These mislabels include semantically equivalent changes, directory or file renames, and initial code importing changes [9–11]. An evaluation framework exists to evaluate the various implementations of the SZZ approach that attempt to remedy these issues [10]. However, the SZZ evaluation framework and the existing SZZ approaches concentrate on mixed bugs or functional bugs, without verifying the validity of the approach on non-functional bugs.

Functional changes and their software fixes are mainly localized, while non-functional bugs may be scattered and require fixes in various parts of the software [12]. For example, if a code change introduces a security vulnerability, security measures to counteract this may be implemented elsewhere [13–15]. If a code change introduces a performance issue, this performance issue may be fixed and improved in a different part of the system [16, 17], for example by changing configuration parameters. Non-functional bugs can be harder to fix than their functional counterparts. For example, it has been found that developers will often spend more time fixing performance bugs than fixing non-performance bugs [18]. As performance bugs are a type of non-functional bug [19], the SZZ approach would seemingly be useful in helping developers locate where to fix a performance bug in the source code. Due to the differing nature of non-functional bugs and functional bugs [20, 21], it is possible for non-functional bugs to present differently in source code, and therefore have different tooling requirements. Due to the scattered nature of non-functional bugs, we suspect that the SZZ approach might perform worse on non-functional bugs than mixed bugs, namely functional bugs which can have a single concrete inducing commit that can be tracked down through the SZZ approach.

In this paper we seek to determine the usefulness of the SZZ approach in finding the cause of non-functional bugs. This paper provides an evaluation of the SZZ approach with respect to non-functional bugs, while previous studies [3, 6, 22, 23] are evaluated on mixed bugs (both functional and non-functional) without distinction. We leverage the NFBugs dataset as a source of identified non-functional bugs [20].

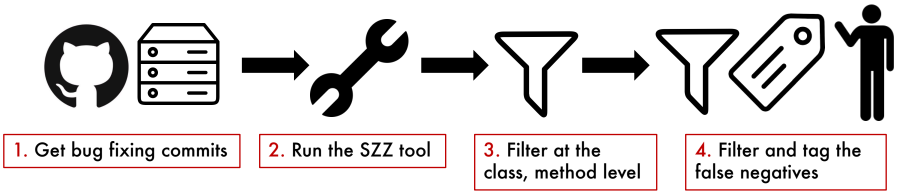

1. Get bug fixing commits
2. Run the SZZ tool
3. Filter at the class, method level
4. Filter and tag the false negatives

Fig. 1: Procedure followed to evaluate bug-inducing commits identified by the SZZ approach

The NFBugs dataset identifies bugs that specifically affect non-functional requirements, as well as the root cause of these bugs, for 65 open-source projects [20]. This dataset presents a vetted source of bugs and their root causes with which to test the SZZ approach. We therefore use this dataset to test the effectiveness of the SZZ approach on non-functional bugs. To determine the usefulness of the SZZ approach with respect to non-functional bugs, we evaluate the SZZ approach based on three criteria: (1) the ability of the SZZ approach to identify bug-inducing changes for non-functional bugs; (2) the differences of inducing changes for non-functional bugs when compared to functional bugs; and (3) the characteristics of bug-inducing commits falsely identified by an SZZ approach as bug-inducing for non-functional bugs.

Upon evaluation of the SZZ approach, we filter out commits wrongly identified as bug-inducting commits by the SZZ approach (false-positives). Furthermore, we conduct a manual verification of the results of SZZ on our dataset to determine whether bug-inducing commits identified by the SZZ approach do in fact cause their related non-functional bugs. We use 132 bug fixes from the NFBugs dataset as our benchmark. For each fixing change, we run the SZZ approach to find how many bug-inducing changes are identified. In total, for all 132 bug fixes, we find that there are a total of 376 candidate bug-inducing commits. We manually observe the commits identified by the SZZ approach to determine whether they are truly bug-inducing. Furthermore, we use the additional insights gained through our manual verification of the SZZ approach to create an extension of the NFBugs dataset. Our findings show that among the 376 identified bug-inducing commits by the SZZ approach, only 79 of them are true positive bug-inducing commits. We manually break down the falsely identified bug-inducing commits into three reasons: multi-purposes bug-fixing commits, 2) bug already being there, and 3) not related to the bug. Our findings show that non-functional bug-inducing commits differ from functional bugs inducing commits, where guidelines to identify falsely detected bug-inducing commits in functional bugs cannot be used reliably when using SZZ to detect non-functional bugs.

The following are the primary contributions of this paper:

- To the best of our knowledge this is the first study to focus exclusively on the use of the SZZ approach to identify the inducing commits of non-functional bugs.
- We manually verify the validity of the SZZ approach on non-functional bugs, and determine potential problems with the approach in dealing with them.

– We augment the NFBugs dataset by including bug fix descriptions that contain the commits where the true bug-inducing changes reside[1].

Paper organization. Section 2 introduces the SZZ approach and other background concepts. Section 3 presents our subject dataset, applications of the SZZ approach, and the steps of our study. Section 4 presents our approach to determine the bug-inducing commits for non-functional bugs. Section 5 presents our empirical study of the SZZ approach when used on non-functional bugs. Section 6 presents a manual investigation of the results of our empirical study, as well as discussion of these results. Section 7 discusses prior work related to the work presented in this paper. Section 8 presents the threats to the validity of our work. Finally, Section 9 concludes the paper.

2 Background

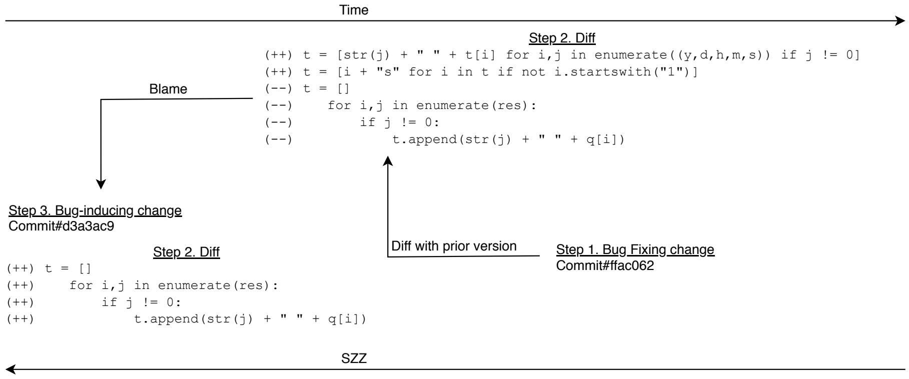
2.1 Overview of the SZZ approach

The SZZ approach was first defined by Sliwerski et al., to identify code changes that induce bugs [7]. Originally, the SZZ approach was used to analyze CVS archives for fix-inducing, or bug-inducing changes [7]: changes that lead to problems. To identify a bug-inducing commit, the SZZ approach requires a code-change that fixes a bug. These code changes are also known as bug-fixing changes. Sliwerski et al. used the SZZ approach to automatically locate fix-inducing changes by linking a code versioning archive to a bug database [7].

The SZZ approach is used to identify the changes that introduce bugs. The approach starts from a bug-fixing change, i.e., a change that is known to have fixed a bug [10]. For each identified bug-fixing change, the SZZ approach analyzes the lines of code that were updated to introduce the fix. In order to identify the change that originally introduced the bug, the SZZ approach traces through the history of the source code management system [10]. The *git annotate* function, now replaced by *git blame* [24] that is provided by most SCM systems is used by the approach to identify the last time a given line of code was changed before the bug-fixing commit [10]. Figure 2 shows a bug fix and a corresponding identified bug-inducing change from the SZZ approach.

Because no SZZ approach provides perfect precision and recall of bug-inducing commits, it is necessary to understand the nature of the results given by the SZZ approach. In an ideal scenario, the SZZ approach can identify the exact changes that introduce a bug. In this ideal case, we consider the results to be changes identified by the SZZ approach that are truly bug-inducing changes, i.e. true positives. These results can directly be used by developers to find the root cause of a known bug. However, if the SZZ approach identifies changes that are not truly bug-inducing changes as bug-inducing, we consider those to be false positives. False positives can cause developers to needlessly look at faultless code. False negatives are truly bug-inducing changes that were missed by the SZZ approach. False negatives would require different tools or manual investigation to find the root cause

[1] Our extension of the NFBugs dataset is publicly available and can be found at: https://github.com/senseconcordia/NFBugsExtended



```
(++) t = [str(j) + " " + t[i] for i,j in enumerate((y,d,h,m,s)) if j != 0]
(++) t = [i + "s" for i in t if not i.startswith("1")]
(--) t = []
(--)    for i,j in enumerate(res):
(--)        if j != 0:
(--)            t.append(str(j) + " " + q[i])
```

```
(++) t = []
(++)    for i,j in enumerate(res):
(++)        if j != 0:
(++)            t.append(str(j) + " " + q[i])
```

Fig. 2: Overview of the SZZ approach. The SZZ approach first looks at the changes made in a bug-fixing change (Step 1). It then uses *git diff* to localize the exact fix (Step 2). Finally, the deletions are traced back to the origin of the deleted code (Step 3). The origin of the deleted code is a potential bug-inducing change.

of a bug. Various modifications of the SZZ approach attempt to improve its true positive rate and reduce its false positives and false negative rates.

2.2 Applications of the SZZ approach

Kamei et al. [25] study defect prediction models that focus on identifying defect-prone software change level, rather than file or package level, referred to as “Just-In-Time Quality Assurance”, where developers can review and test these risky changes while they are still fresh in their minds. Kamei et al. [25] use the SZZ approach to determine whether a change introduces a defect. In this case, the SZZ approach is used to link each defect fix to the source code change introducing the original defect by combining information from the version archive with the bug tracking system. Findings from Kamei et al. [25] indicate that “Just-In-Time Quality Assurance” may provide an efficient way to focus on the most risky changes and thus reduce the costs of developing high-quality software.

McIntosh et al. [26] study the effectiveness of JIT defect prediction models as systems evolve. Through a longitudinal case study of open source systems, they find that fluctuations in the properties of fix-inducing changes can impact the performance and interpretation of JIT models. They detect whether a change is potentially fix-inducing using the SZZ approach. McIntosh et al. [26] find that the discriminatory power (AUC) and calibration (Brier) scores of JIT models drop considerably one year after being trained. While McIntosh et al. [26] use the data from the SZZ approach to predict future bug-inducing changes in JIT models and evaluating those models, our paper focuses on the data itself that is produced by the SZZ approach, i.e., the bug-inducing changes detected by the SZZ approach.

Current adoption of techniques that predict software quality remains low. One of the reasons for the low adoption rate of current analytics and prediction techniques is the lack of actionable and publicly available tools. Rosen et al. [27] present Commit Guru [27], a publicly available, language agnostic, analytics and prediction tool that identifies and predicts risky software commits mined from any Git repository. Additionally, Commit Guru [27] automatically identifies risky (i.e., bug-inducing) changes and builds a prediction model to assess the likelihood of a recent commit being bug-inducing in the future. A similar approach to the SZZ approach is used to determine bug-fixing commits [27].

The wide application of the SZZ approach motivates our research of the usefulness of the SZZ approach when used to identify the inducing changes on non-functional bugs.

3 Study design

In this section, we present the design of our exploratory study. We first present the dataset as the subject of our study. Afterwards, we layout the two steps of our study and their motivations.

3.1 Subject dataset

Because SZZ requires known bugs as inputs, our research hinges on the availability of vetted non-functional bugs. Datasets of non-functional bugs have been produced and vetted in prior research [20,28]. However, non-functional bugs and their causes, sometimes require domain knowledge to be understood and detected. It is therefore beneficial for the root-causes of each non-functional bug to be clearly indicated as provided by the NFBugs dataset [20]. Such information is crucial in our study to verify the bug-inducing changes identified by the SZZ approach.

Fortunately, recent research by Radu and Nadi [20] initiated an open repository that contains a dataset of real-world non-functional bugs, with each bug's detailed information (see Listing 1). The NFBugs dataset contains bugs from 65 open source GitHub projects: 40 Java projects and 25 Python projects. These projects contain 89 listed Java non-functional bugs and 43 listed Python non-functional bugs. For each project, NFBugs lists at least one bug, its respective fix and its detailed root causes. Each listed bug has been manually identified and has a corresponding YAML file, with the file and method that pinpoints the bug as well as a short description of the bug, however it contains no mention of commits that induced the bug. An example of a YAML file is shown in Listing 1. This study makes use of the NFBugs dataset to advance the state-of-the-art and allow future replication.

```
source:
    name: github-search
project:
    name: VS_test
    url: https://github,com/georgeriz/VS_test/
fix:
    tag: performance
    description: Replacing .append loop with list comp improves
        performance because it does not have to access the
        instance method
    commit message: > project duration:list comprehension
        instead of for loop
    commit: https://github,com/georgeriz/VS_test/commit/ffac062
location:
    file: duration/duration.py
    method: duration(seconds)
api: builtins.list
api change:
    builtins.list.append -> builtins.list list comprehension
rule: use list comprehension instead of list.append loops to
    create lists efficiently
```

Listing 1: An example YAML file from the NFBugs dataset for a non-functional bug **ffac062** in VS_test

3.2 Implementation of SZZ

Since its creation, SZZ has been modified and re-implemented with various modifications [6, 7, 10, 29]. In this paper we concentrate on the MA-SZZ (meta-change aware SZZ) implementation of SZZ [10]. Meta-changes are source code independent changes; source code management branch changes, source code merges, and changes to file properties such as end-of-line changes are all examples of meta-changes [10]. We concentrate on the MA-SZZ implementation of SZZ because it is commonly used in prior research [10] and is language agnostic. The MA-SZZ [10] approach is an implementation of SZZ that adds meta-change awareness to the AG-SZZ approach [6]. MA-SZZ uses an annotation-graph to represent the evolution of each line of code within source files. Depth-first search is performed on the annotation-graph to find potential bug-inducing changes. We use our own implementation of the MA-SZZ approach to perform our evaluation of the SZZ approach on non-functional bugs. Because prior research has used MA-SZZ to observe functional bugs, we also use MA-SZZ to observe non-functional bugs. We did not want to introduce a different SZZ approach (e.g., RA-SZZ [11]) because although RA-SZZ has shown improvements over other SZZ approaches [11], changing the approach may introduce confounding factors and make the results impossible to compare fairly. However, while we suspect that refactorings actually can have an effect on non-functional bugs, as they do on functional bugs, we have not yet found a study that presents the effects of refactoring on non-functional software bug incidence detection. Therefore, to be conservative, we use an MA-SZZ implementation of the SZZ approach rather than an RA-SZZ implementation because using RA-SZZ would potentially change the performance of the method and make the results difficult or impossible to compare with prior studies.

The SZZ approach requires a bug-fixing change as an input. The approach then performs its depth-first search to find potential bug-inducing changes. Therefore, we require bug-fixing changes, specifically bug-fixing commits, to use as inputs for the SZZ approach. The NFBugs dataset lists the bug-fixing commit hash of each non-functional bug instance. Most of the closed bug reports identify the fix commit that closed the bug report. We therefore use these bug-fix commits as input data for our study.

3.3 Steps of our study

The steps of our study can be found in Figure 1. Using the non-functional bug dataset and the MA-SZZ implementation of SZZ, we first run the SZZ approach to identify bug-inducing commits. Then we carry out our study in two steps.

Step 1: Manually verifying the bug-inducing commits identified by SZZ

Prior studies have manually evaluated the results that are generated by the SZZ approach [10]. However, those studies do not make a distinction between functional and non-functional bugs during their evaluation. Nonetheless, it has been shown that non-functional bugs present different characteristics than functional bugs [16]. In particular, non-functional requirements describe the quality attributes of a program, as opposed to its functionality [30]. Therefore, the prior manual evaluation results for SZZ approaches may not generalize to non-functional bugs.

In addition, in order to further improve the SZZ approach on non-functional bugs, it is necessary to first obtain a dataset of non-functional bugs with correctly identified bug-inducing commits. The correct bug-inducing commits are paramount for any further analysis. In this step, we therefore seek to manually verify the inducing changes for non-functional bugs and complement the existing dataset of these non-functional bugs by incorporating their corresponding true inducing changes.

Step 2: Automatically evaluating the bug-inducing commits identified by SZZ

Manually evaluating results from the SZZ approach is time consuming and almost impossible to scale in practice. However, practitioners may always face the challenge of having falsely identified bug-inducing changes from SZZ approaches. To address such a challenge, prior work by Costa et al. [10] provides characterizations of SZZ results using three characteristics of bug-inducing changes as guidelines: 1) *Earliest bug appearance*, 2) *Future impact of a change*, and 3) *Realism of bug introduction.* These bug characteristics can be used to provide a fine-grained evaluation of the SZZ approach. *Earliest bug appearance* measures when a bug was introduced. *Future impact of a change*, analyzes the number of future bugs that a given bug-inducing change introduces. Finally, *realism of bug introduction* analyzes whether the bug-inducing changes found by SZZ approaches realistically correspond to the actual bug introduction context. If these automated guidelines cannot reliably identify the falsely identified bug-inducing commits for non-functional bugs, practitioners and researchers may not adopt these guidelines for evaluating the results of SZZ approaches on other datasets.

The results of **Step 1: Manually verifying the bug-inducing commits identified by SZZ** are used in Section 4. The results of **Step 2: Automatically evaluating the bug-inducing commits identified by SZZ** are used in Section 5.

4 Manually verifying bug-inducing commits for non-functional bugs identified by the SZZ approach

In this step, we manually verify the bug-inducing commits that are automatically identified by the SZZ approach.

Before applying the SZZ approach on our dataset, we first exclude nine bugs (five from Java and four from Python) where the bug-fix commits are part of merge commits. We exclude merge commits since studies shows that the SZZ approach should not take merge commits into account due to the noise that can be introduced by a code merge [9].

Afterwards, we run the SZZ approach for the remaining 123 bugs. The SZZ approach produces a list of bug-inducing candidate commits for each bug. After running the SZZ approach on the remaining bugs, we obtained a total of 284 candidate bug-inducing commits for Java and 92 for Python. Each bug in the NFBugs database has been manually identified and has a corresponding YAML file listing the file(s) and method(s) for the bug as well as a short description of the bug, as shown in Listing 1. Each of these remaining candidate bug-inducing commits, are manually verified by three of the authors of this paper independently, to avoid introducing any bias.

The steps performed in this paper for the manual analysis of bug-inducing commits are as follows:

- **Step A**: The reviewers read the description of a non-functional bug from the NFBugs dataset.
- **Step B**: The reviewers examine the code from the mentioned bug-fix commit.
- **Step C**: For each of the candidates identified as bug-inducing commits by the SZZ approach, the reviewers examine the code that is changed in the commit and determine whether it induces the corresponding bug.

After all steps are individually completed by the first, third, and fourth authors for all bugs remaining in the dataset, the reviewers then meet to discuss disagreements. All three reviewers must have the same classification (i.e., bug-inducing or not bug-inducing) for a candidate commit, otherwise this is marked as a disagreement. The disagreements are resolved as follows:

- **Step D**: The reviewers re-read the bug-fix and the bug-inducing commit in question.
- **Step E**: Each reviewer states the reason why they think the identified commit is bug-inducing or not bug-inducing.
- **Step F**: The reviewers discuss until all three agree on a final decision.

All agreements and disagreements are recorded and used to calculate the Multi Kappa Fleiss score, a robust statistic useful for either interrater or intrarater reliability testing [31]. Afterwards, the three individuals meet and discuss any differences and reach a consensus.

Finally, we manually investigate all the false candidate of bug-inducing commits for non-functional bugs, in order to uncover reasons of such faults.

Results.

For the manual examination of the candidate bug-inducing commits, there were a total of 27 candidate bug-inducing commits disagreements, 20 from Java and 7 from Python bugs. To quantitatively evaluate how often we agreed during manual

Table 1: Java and Python Projects: True-Positives and False-Positive after filtering based on relevant method and class based on the description provided

Language	No. Pairs of identified bug-inducing commits	TP	FP
Java	284	109	175
Python	92	50	42

evaluation, we use the Multi Kappa Fleiss score [31]. The Multi Kappa Fleiss score was 0.728 - a moderate level of agreement for Java, and 0.815 - a strong level of agreement for Python [31]. All 27 candidate bug-inducing commit disagreements were resolved through discussion between the three individuals. In many cases, the disagreements split 1:2 were resolved due to one individual missing some critical information while manually reviewing the code. After a second look at the code, the reviewers' response allowed the three reviewers to reach consensus.

Only 41 out of 123 bugs have fully correct bug-inducing changes identified by the SZZ approach. 27 bugs have fully wrong identified bug-inducing changes. **For the bugs where the SZZ approach was not able to identify any truly bug inducing commits, we manually look in the repository to find the commits that were bug-inducing for bugs: accounting for 27 out of 123 bugs.** 19 bugs have a combination of correct bug-inducing changes and incorrect bug-inducing changes, e.g., in the case of multi-purpose bug-fixing commits, some code changes in the commit are not done to fix the bug, tracking such changes may result in falsely identified bug-inducing commits.

Out of the 376 bug-inducing commits, 217 commits were ruled out as false positive bug-inducing commits which were not in the same method or class from the total 376 identified bug-inducing commits. For the remaining 159 bug-inducing commits, there are a total of 80 bug-inducing commits that are false positives and therefore were falsely labelled as bug-inducing commits by the SZZ approach shown in Figure 3. Based on our findings on the NFBugs dataset, there are 55 performance bugs that have 94 bug-inducing commits with 45 truly bug-inducing commits, and there are five security bugs that have five bug-inducing commits with four truly bug-inducing commits.

Prior studies have reported that the SZZ approach still needs improvements to accurately identify bug-inducing changes [10]. For MA-SZZ, Costa et al. [10] report a 0% to 17% disagreement ratio, where they count a bug as a disagreement if all of the candidate bug-inducing changes identified by the SZZ approach for that bug are classified as incorrect. However, based on our manual analysis results, the SZZ approach performs even worse on non-functional bugs.

Our findings show that among the 376 identified bug-inducing commits for 123 bugs, only 79 true positive bug-inducing commits are identified by the SZZ approach. Only 40 bugs have fully correctly identified bug-inducing changes.

We manually identify three reasons that account for all of falsely identified bug-inducing commits for non-functional bugs: 1) multi-purposes bug-fixing commits, 2) bug already being there, 3) not related to the bug shown in Table 2 and Figure 3. Reasons for not related to the bug include modifications of Javadoc or comments, the additional or removal of Java modifiers to variables, reverting changes, and

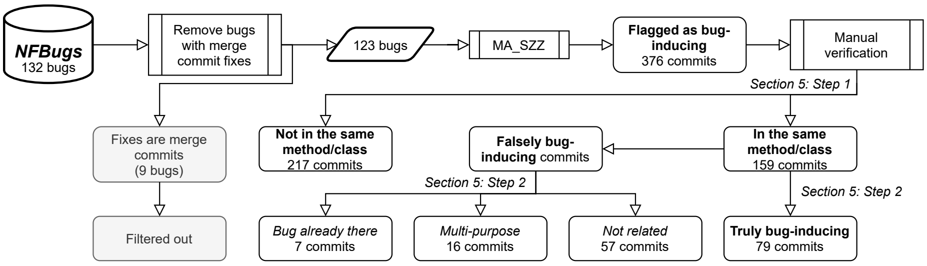


Fig. 3: Breakdown of false positive bug-inducing commits which were not in the same method/class mentioned from the bug-fix description from NFBugs.

some where we cannot find overlapping lines between removed lines in the fix commit and added lines in the identified bug-inducing commit.

Table 2: Breakdown of falsely identified non-functional bug-inducing commits into multi-purposes bug-fixing commits, 2) bug already being there, 3) not related to the bug.

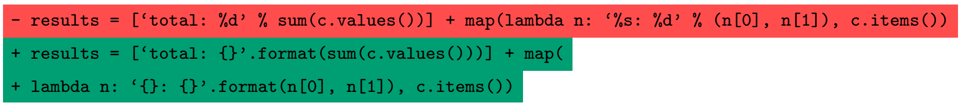
	Total	Multi-purpose	Bug already being there	Not related to the bug
Java	54	8	7	39
Python	26	8	0	18

Multi-purposes bug-fixing commits. Our SZZ approach tracks all the code changes in the bug-fixing commits to identify bug-inducing commits. However, if a bug-fixing commit has multiple purposes, (i.e., some code changes in the commit are not done to fix the bug), tracking such changes would result in falsely identified bug-inducing commits. Unfortunately, we find a large number of cases where the bug-fixing commits are not dedicated to fixing a non-functional bug. For example, the commit message of commit **2391544** from the *Catacomb-Snatch* (in Java) project is "*Code Cleanup: Closed resource leaks, and removed or commented out unused code/resources, and did some code layout clean up (braces on ifs and correct indentation)*". The first part of the commit message is clearly related to the non-functional bug of system resource leaks; while on the other hand, removing the unused code and the code layout cleanup may introduce noise to the SZZ approach.

In total, eight out of 54 falsely identified bug-inducing commits in Java and eight out of 26 in Python are due to multi-purpose bug-fixing commits. We would like to further verify whether these falsely identified bug-inducing commits are from a small number of bugs that are fixed in a multi-purpose manner. We find that these falsely identified bug-inducing commits originated from only five bugs in Java and one bug Python. Since there exists five bugs in Java and two bugs in Python with multi-purpose fixes, that have all true-positive identified bug-inducing commits. By manually looking at these bug-fixing commits, we find that all of the changes that are not associated with bug fixing are adding lines of source code, not affecting results of SZZ approaches. On the other hand, since the SZZ approach has become an application of automated bug-detection tools, multi-purpose bug-fixing commits have become a shortcoming of automatically applying SZZ for other

Table 3: Number of single-purpose and multi-purpose bugs with false positive and all true positive bug-inducing changes.

	Single-purpose		Multi-purpose	
Language	with FP	with all TP	with FP	with all TP
Java	23	25	5	5
Python	17	8	1	2



```
- results = [‘total: %d’ % sum(c.values())] + map(lambda n: ‘%s: %d’ % (n[0], n[1]), c.items())
+ results = [‘total: {}’.format(sum(c.values()))] + map(
+ lambda n: ‘{}: {}’.format(n[0], n[1]), c.items())
```

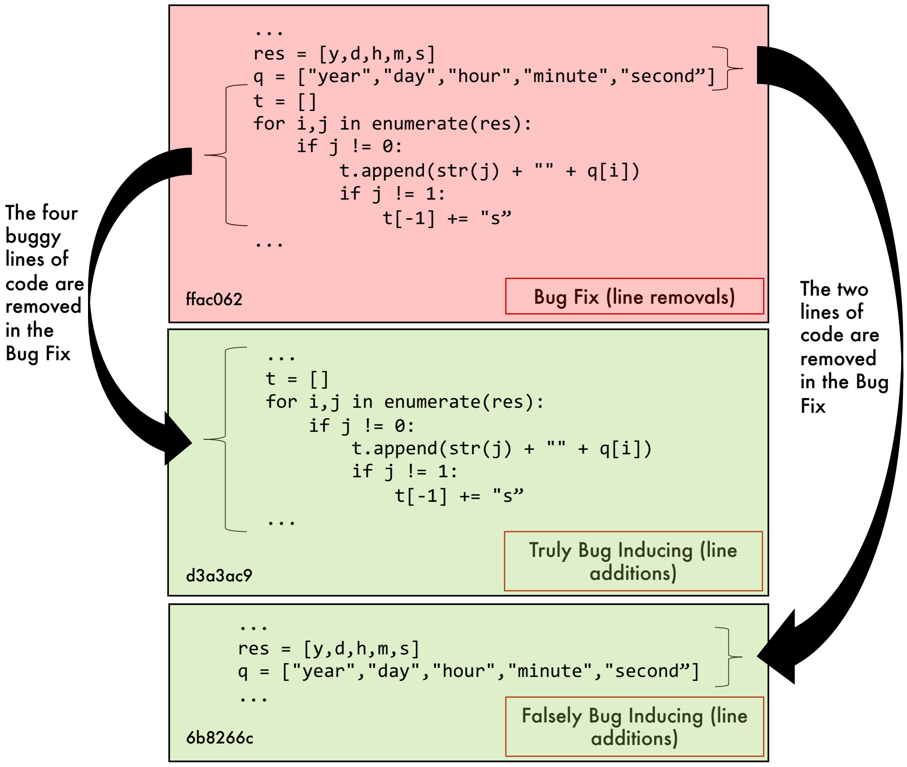
Fig. 4: Simplified example of an "Already there/Re-factoring" bug-inducing code line that was modified to semantically equivalent code

downstream tasks, such as Just-In-Time Quality Assurance [3]. Our results can be used to understand how badly the input noise actually affects the results of the SZZ approach and later impact its downstream tasks.

We also want to see whether commits of single-purpose or multi-purpose affect the performance of the SZZ approach in identifying bug-inducing commits. To do this, we also breakdown the single-purpose fixes based on whether the identified bug-inducing commits are false positive-bug-inducing commits or true positive-bug-inducing commits in Table 3. 23 out of 48 and 17 out of 25 single-purpose bug fixing commits, for Java and Python, respectively, lead to false positives. The results are comparable with the ones with multi-purpose bug fixing commits.

Bug already being there. For seven out of 54 falsely identified bug-inducing commits in Java and no cases in Python, when we examine the bug-inducing commit, we find that the non-functional bug already exists. Such a finding shows that in many cases, after the non-functional bugs are induced, developers may change the same lines of code, while not realising there exists a non-functional bug. In some other cases, the developers refactor or reformat the same line of code without actually changing the functionality. In either scenario, the SZZ approach may consider the later changes as bug-inducing instead of the original changes. This phenomenon is intuitive since non-functional bugs often take a long time to be discovered and fixed [16]. Therefore, considering the most recent code change before the bug reporting date may not be a suitable heuristic for non-functional bugs. We did not find any cases of bugs already being there in our Python dataset.

We present an example of a semantically equivalent change wrongly identified as a bug-inducing change in Figure 4. The bug-inducing code lines that the SZZ approach looks for are **results = [‘total: ’.format(sum(c.values()))] + map(** and **lambda n: ‘{}: {}’.format(n[0], n[1]), c.items()**). These lines are shown as additions in this commit in green, however, in this same commit, the lines are also shown as a removal in red. The difference between these two is the string formatting, no logic is altered. The commit message is: "**String formatting**". The SZZ approach stops at this commit and flags it as bug-inducing, since the bug-inducing code has technically been added in this line. However, although this is technically correct, this code was actually first introduced further back in time with different formatting in a different commit. The approach wrongly suspects



```
...
res = [y,d,h,m,s]
q = ["year","day","hour","minute","second”]
t = []
for i,j in enumerate(res):
    if j != 0:
        t.append(str(j) + "" + q[i])
        if j != 1:
            t[-1] += "s”
...
```

ffac062
Bug Fix (line removals)

```
...
t = []
for i,j in enumerate(res):
    if j != 0:
        t.append(str(j) + "" + q[i])
        if j != 1:
            t[-1] += "s”
...
```

d3a3ac9
Truly Bug Inducing (line additions)

```
...
res = [y,d,h,m,s]
q = ["year","day","hour","minute","second”]
...
```

6b8266c
Falsely Bug Inducing (line additions)
The four buggy lines of code are removed in the Bug Fix
The two lines of code are removed in the Bug Fix

Fig. 5: Example of a bug-fix (ffac062) with two inducing commits, one bug-inducing (d3a3ac9), and one falsely bug-inducing (6b8266c)

the beautifying commit as a bug-inducing commit, which we describe as a case of “Bug already being there”.

Not related to the bug. For Java we find 39 out of 54 falsely identified bug-inducing commits in Java and 18 out of 26 in Python that are not related to the non-functional bug. Figure 5 shows two candidate commit bugs for a bug from the NFBugs dataset. The corresponding YAML file is shown in Listing 1. Upon manual analysis by the three reviewers, commit **6b8266c** in Figure 5 is identified as a false positive bug-inducing commit whereas commit **d3a3ac9** in Figure 5 is identified as a true positive bug-inducing commit. Commit **6b8266c** in Figure 5 was ruled out because it does not have a relation to the **builtins.list.append** and **use list comprehension instead of list.append loops to create lists efficiently** as stated in the Listing 1 bug description. From the description, the reviewers knew to look for a commit that refers to **builtins.list.append**.

We manually identify three reasons for falsely identified bug-inducing commits for non-functional bugs including 1) multi-purposes bug-fixing commits, 2) bug already being there, and 3) changes not related to the bug.

5 Automatically evaluating bug-inducing commits identified by the SZZ approach

In this step, we apply the three automated guidelines that are proposed by Costa et al. [10], i.e., 1) *Earliest bug appearance*, 2) *Future impact of a change*, and 3) *Realism of bug introduction*, in order to study whether such automated guidelines can help identify the falsely detected bug-inducing commits by the SZZ approach when used on non-functional bugs. The results are summarized in Table 4.

– *Earliest bug appearance.* For the corresponding candidate bug-inducing changes of each bug, we check the time of the change and the impacted version of the software. If the impacted version of the software is earlier than the candidate bug-inducing change, the candidate bug-inducing change is considered false.
– *Future impact of a change.* For each bug-inducing change, we calculate the count of induced bugs and the time-span of the induced bugs. If one change induces too many bugs or the induced bugs are across a long period of time, the bug-inducing change may be false.
– *Realism of bug introduction.* For each bug, we calculate the time-spans between the bug-inducing changes for each bug. If the time-span is too long, the bug-inducing changes may be false. We also consider bug-inducing changes after a bug-fixing change.

We use the results of the guidelines presented above for all bugs (functional and non-functional) as a baseline. For each of the two languages in NFBugs: Java and Python, we pick the top two projects with the largest number of reported bugs in the NFBugs dataset. We pick the top two projects since the project with the third highest number of reported bugs had significantly fewer reported bugs compared to the top two, making the sample size too small for comparison. In particular, we focus on four projects that are included in the NFBugs dataset, i.e, *Elasticsearch* and *Jenkins* (in Java) and *Falcon* and *Gae-boilerplate* (in Python). We choose these four projects since they contribute a large number of bugs for their respective language in the NFBugs dataset. We first extract all bugs (both functional and non-functional). For *Falcon* and *Jenkins* we look at their JIRA bug reports, meanwhile the other projects rely on GitHub's issue tracker. All of the issue trackers in our data are open to access. In order to study the functional and non-functional bugs that are around the same time period during development, we extract all the commits that are in a time period that is between six months before the reporting date of the first non-functionally bug and six months after the date of the last reported non-functional bug. We use the issue id (e.g., # with a number in GitHub issue tracker) in the commit message to link each issue and its issue fixing commit. We only consider the issues that are tagged with a *bug* label in either the GitHub issue tracker or JIRA. We assessed the functional-bug fixes in these four projects in the dataset, using a search for the issue id in the commit messages to identify the bug fixing commits, and then we further identify the bug-inducing commits related to them by running the SZZ approach. For the manual analysis of functional bugs, we pick a random sample with a confidence interval of 95% ± 10% and end up with 96 identified bug-inducing commits in Java, and 91 identified bug-inducing commits in Python. We execute the MA-SZZ approach on the non-functional and functional bugs. Afterwards, we evaluate the three criteria: earliest bug appearance, future impact of a change, and realism of bug introduction

Table 4: Percentages of bugs that will be identified by the three guidelines proposed by Costa et al. [10]

	Reported by the prior study	Reported by NFBugs
Future impact (commit level)	95%	54%
Earliest bug appearance (bug level)	0-3%	0-17%
Realism of a bug (bug level)	3%	46%

using scripts that we created to calculate these guidelines proposed by Costa et al. [10].

Finally, we calculate the metrics that correspond to the three guidelines used by Costa et al. [10], i.e., 1) *Earliest bug appearance*, 2) *Future impact of a change*, and 3) *Realism of bug introduction*.

Results. We summarize the breakdown of results based on the three guidelines proposed Costa et al.'s [10] prior work, for identifying falsely detected bug-inducing commits by the SZZ approach. We compare our results to Costa et al.'s [10] in Table 4.

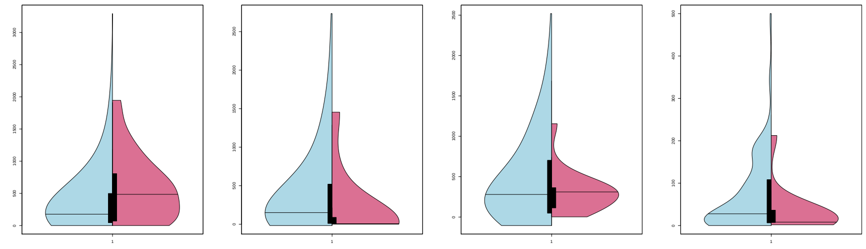
***Earliest bug appearance* and *Realism of bug introduction* [10] are not effective in identifying false candidate of inducing changes of non-functional bugs from SZZ approaches.** We observed only one case of *Earliest bug appearance* where there is a bug-inducing change for which the bug report for the corresponding fix was made even before the bug-inducing change date. This is unrealistic as it is not possible to report a bug in the software before the bug was introduced.

In their study, Costa et al. [10] uncovered some unrealistic changes. Costa et al. [10] found that 46% of bugs are caused by bug-inducing changes that span at least one year. It is unlikely that 46% of all bugs were caused by code changes that are years apart. We identify another type of unrealistic result in our own findings where a bug-inducing change was fixed or reported even before the bug change commit time. However, we only observed one case of *Realism of bug introduction* where for a bug, the time span between the bug-inducing changes is too long.

Therefore, since the *Earliest bug appearance* and *Realism of bug introduction* guidelines from Costa et al. [10] occur infrequently in the NFBugs dataset, we cannot reliably use them to identify false bug-inducing commit candidates.

***Future impact of a change* can be used as an indicator to identify false candidate of inducing changes of non-functional bugs from SZZ approaches.** We observed 36 cases of *Future impact of a change*, where one change induced many bugs or induced bugs across a long period of time. Part of Costa et al.'s [10] findings show that 29% of the bug-inducing changes lead to multiple future bugs that span at least one year. This suggests that SZZ approaches still lack mechanisms to accurately flag bug-inducing changes as it is unlikely that all 29% of the bug-inducing changes in a project introduce bugs that took years to be discovered.

Functional and non-functional bug-inducing commits do not overlap. We find that there are no cases within the four projects that we examined where a bug-inducing commit for a non-functional bug had also induced a functional bug. This indicates that non-functional bugs and functional bugs are quite different in nature. Non-functional bugs have may require different tooling requirements if they manifest differently in the source code, and are scattered across and require fixes in various parts of the software.



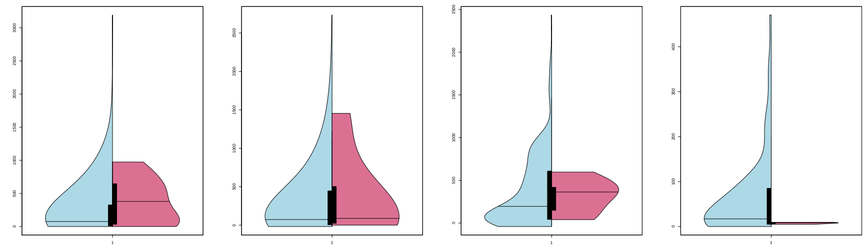
(a) Elasticsearch
p-value: 0.004

(b) Jenkins
p-value: 0.112

(c) Falcon
p-value: 0.796

(d) Gae-boilerplate
p-value: 0.218

Fig. 6: Comparison of Functional (left) and Non-Functional (right) bugs for days between inducing changes and fixing changes.



(a) Elasticsearch
p-value: 0.698

(b) Jenkins
p-value: 0.228

(c) Falcon
p-value: 0.796

(d) Gae-boilerplate
p-value: 0.390

Fig. 7: Comparison of Functional (left) and Non-Functional (right) bugs for days between earliest inducing changes and fixing changes.

Based on Figure 6 and Figure 7 we can see that the median non-functional bugs appear to take a longer time on average to fix, compared to functional bugs. However, we cannot make any statistically significant conclusions due to the insufficient sample size. We use the data presented in violin plots in Figure 6 and Figure 7, and perform Wilcoxon rank sum tests, comparing the non-functional and functional bugs. Elasticsearch's comparison for functional bugs and non-functional bugs of days between all bug-inducing changes for fixing changes is the only statistically significant comparison (p=0.004).

The Cliff's Delta value for Elasticsearch's comparison for functional bugs and non-functional bugs of days between all bug-inducing changes for fixing changes is: -0.251. Cliff's Delta indicates that the type of bug (Functional or Non-Functional) has a small effect on the number of days between inducing changes and fixing changes for the Elasticsearch project.

The false results of SZZ on functional and non-functional bugs may be different. We further perform manual analysis on the functional bugs from Elasticsearch, Jenkins, Falcon, and Gae-boilerplate. Upon performing manual analysis on the functional identified bug-inducing commits following the steps listed in Section 4 we have calculated the Multi Kappa Fleiss score to be 0.736 - a moderate level of agreement for Java, and 0.822 - a substantial level of agreement for Python. Through our validation of the functional bugs, we further discarded three bugs from the Python projects and six bugs from the Java projects, because they were non-functional bugs, rather than functional bugs. For Python, out 56 of the examined bugs, 40 of them have at least one truly identified bug-inducing commit. For Java, out of 76 of the examined bugs, 31 of them have at least one truly identified bug-inducing commit. 50 out of the 88 identified bug-inducing commits were not truly bug-inducing for Python, and 59 out 89 identified bug-inducing commits in Java were not truly bug-inducing. In summation, 61.7% of the functional commits identified by the SZZ approach were not truly bug inducing compared to 80.0% for non-functional bugs. Similarly to Table 2, we break down the falsely identified functional bug-inducing commits in Table 5. Through observation, non-functional bugs experience more false positives bug-inducing commits because of multi-purposes bug-fixing commits. While it is possible for commits to be falsely identified due to multi-purpose bug-fixing commits in functional bugs, this occurs more rarely in functional bugs than in non-functional bugs. The bug already there reason for false positive bug incidence detection does appear in functional and non-functional bugs, however, they appear to experience different magnitudes of this fault: 20.2% for functional bugs, and 8.7% for non-functional bugs. Therefore, our results show that **not only is functional bug incidence detection more accurate than non-functional bug incidence detection, but functional bug false positive incidences also present themselves differently than for non-functional bugs**.

Table 5: Breakdown of falsely identified functional bug-inducing commits into multi-purposes bug-fixing commits, 2) bug already being there, 3) not related to the bug.

	Total	Multi-purpose	Bug already being there	Not related to the bug
Java	59	2	14	43
Python	50	0	8	42

Non-functional bug-inducing commits differ from functional bugs inducing commits. Guidelines to identify falsely detected bug-inducing commits by the SZZ approach such as Earliest bug appearance and Realism of bug introduction cannot be reliably used when using SZZ to detect non-functional bugs.

6 Discussion

Prior studies found that SZZ implementations still have room for improvement and suggest handling special cases to more accurately detect bug-inducing changes [6, 10]. The three cases of improvement include handling are:

- *Semantically equivalent changes*: Some implementations of the SZZ approach, including the one used for the purpose of the paper, take into account comments, blank lines, indentation and white-space changes. However, even adjusted SZZ approaches, still have problems with other types of format changes such as reordering and renaming parameters [32].

- *Directory or file re-names*: the SZZ approach cannot flag potential bug-inducing changes that are actually directory/file renaming changes since version control systems may not accurately track renamed files. Therefore, the SZZ approach cannot connect code changes that are performed on older versions of a renamed file [10]. Broken historical links could be recovered heuristically by using repository mining techniques. [33]
- *Initial code important changes*: In cases where a project has been migrated from one version control system to another, SZZ approaches should trace back into the old version control system data since in those cases the initial commit of an version control system may not be the actual starting point of the project. However, current SZZ approaches are not able to trace back across different version control system and therefore cannot handle imported changes from a change in version control system [10].

We examine whether the above three improvements would help improve identifying the true bug-inducing commits for non-functional bugs. In order to perform such examination, we need to have all of the true bug-inducing commits for all of the non-functional bugs in NFBugs. Therefore, we examine the 28 non-functional bugs that do not have true bug-inducing commits detected from the first step of our study (cf. Section 5). We manually check the history of the source code in each project and try to identify the true bug-inducing commits for each of those bugs. In particular, the first author of the paper proposed true bug-inducing commits for each bug and the third and fourth authors independently verified the proposed bug-inducing commits. If at least one author does not agree about a proposed bug-inducing commit, the three authors conduct further discussion about the specific case. If a proposed bug-inducing commit was deemed to be false, the first author did another round of examination to propose other commits as bug-inducing. This procedure was repeated until all three authors agreed that the bug-inducing commit was correctly identified.

We find that 60 out of 82 of the true bug-inducing commits for the non-functional bugs in Java and 21 out of 32 in Python, are actually the initial version of the corresponding code snippet. In other words, **the non-functional bugs were induced when developers first introduced the code into the project.**

Finally, we manually examine whether the above three improvements would help detect the true bug-inducing commits as bug-inducing. We only find two cases where addressing semantically equivalent cases can avoid mistakes, while the other two improvements do not have *any* impact on the results. We believe that this is the case since our subject systems all use Git as version control systems, where directory or file re-names are handled by the version control systems. In addition, the subject dataset may not have many cases of migrating version control systems. Therefore, the improvements proposed by prior studies may not help address the falsely detected bug-inducing commits for the non-functional bugs in NFBugs. However, they may indeed help when if the non-functional bugs are from a project that uses an older version control system (like Subversion) and/or has gone through migrations from one version control system to another.

7 Related Work

In this section, we situate our work within the context of past studies that have evaluated SZZ approaches and studying non-functional bugs. In addition, we present a comparison between our findings and the ones from related work in Table 6.

7.1 Implementations of SZZ

The first SZZ approach was defined by Sliwerski et al. [7], to identify the changes that introduce bugs. SZZ begins with a bug-fixing change, i.e., a change that is known to have fixed a bug. B-SZZ (the basic SZZ implementation) has several limitations, as it may flag style changes [6], which do not affect the system. Since then, there has been several improvements: instead, Kim et al. [6] proposed an SZZ implementation that excludes style changes from the analyses. Furthermore, Kim et al. [6] propose the use of the annotation graph. We refer to the SZZ implementation that is proposed by Kim et al. as AG-SZZ.

Costa et al. [10] propose the MA-SZZ implementation, which is built on top of AG-SZZ, however potential bug-introducing changes that are meta-changes are now removed. They find that B-SZZ has the lowest disagreement ratio in general (0%-9%), followed by the MA-SZZ (0%-17%) [10]. The bugs analyzed by MA-SZZ have the shortest time-span of bug-introducing changes (316 days), while B-SZZ has the longest time-span of bug-introducing changes [10]. Costa et al. [10] also report that MA-SZZ returns the second highest count of future bugs and second highest timespan of future bugs, following B-SZZ. Costa et al.'s [10] study aims to evaluate the SZZ approach rather than studying the characteristics of bug-inducing changes that are detected by SZZ. Prior work in Just-In-Time defect prediction uses MA-SZZ to identify bug-inducing changes [3,26] as a ground truth for building the prediction models. Prior work in studying refactoring changes also uses MA-SZZ by incorporating it with RefDiff to propose a refactoring aware SZZ implementation [34].

Borg et al. [24] propose an open implementation of the SZZ approach for git repositories. The authors include a usage example for the Jenkins project and conclude with a case study on JIT bug prediction. The SZZ Unleashed implementation is based on Sliwerski et al.'s [7] work, as well as later enhancements by Williams and Spacco [32]. Because MA-SZZ is also based on Sliwerski et al.'s [7] work, our findings on non-functional bugs may benefit SZZ Unleashed [24] as its purpose is for git repositories, which can contain a mixture of functional and non-functional bugs.

7.2 Evaluating SZZ approaches

Kim et al. present algorithms to identify bug-inducing changes automatically and accurately. They compare their algorithms to the SZZ [6] approach. They removed false positives and false negatives by using annotation graphs, and ignored non-semantic code changes and outlier fixes. They also manually inspected the commits listed as bug fixing to determine if they were indeed changes that fixed a bug in the code. In our paper, we evaluate our dataset with MA-SZZ, while Kim et al. [6] evaluated the approach on the first version of the SZZ approach.

Williams et al. revisit the SZZ approach by outlining several improvements to the approach [32]. They replace annotation graphs with linear number maps to track unique source code lines as they change over software evolution. Their enhanced approach uses weights to map the evolution of a line. They also use DiffJ, a Java syntax-aware diff tool to ignore comments and ignore cosmetic changes [35]. Furthermore, they verify how often bug-inducing changes identified by the SZZ approach are truly bug-inducing changes. We want to compare an improved SZZ approach implementation: MA-SZZ, as the study performed by Williams et al. compared their improvements to the first SZZ approach implementation [7].

Costa et al. [10] introduced a framework to evaluate the results of SZZ approach implementations. They note that little effort has been made to evaluate SZZ's results, despite its role as the foundation of several research areas in software engineering [10]. The framework evaluates the approach with three criteria: the earliest bug appearance, the future impact of changes, and the realism of bug introduction [10]. The framework is evaluated on five SZZ implementations using data from ten open source projects. Their findings show that previous proposed improvements to SZZ approaches tend to inflate the number of false positive bug-inducing changes. A single bug-inducing change may be blamed for introducing hundreds of future bugs and SZZ implementations report that at least 46% of the bugs are caused by bug-inducing changes that are years apart from one another [10]. Our study builds on the work from Costa et al. by using their evaluation criteria as well as new evaluation criteria to evaluate SZZ approaches on non-functional bugs rather than on a mixed dataset containing both functional and non-functional bugs. Similarly to Costa et al. [10], in Figure 7 we evaluate our data on the earliest bug appearance, in Figure 6 we evaluate our data on the future impact of changes.

Fan et al. [9] studied the impact of mislabelled changes of the SZZ approach on JIT prediction. They analyze four different SZZ implementations and build the JIT prediction models using the labeled data of these four variants [9]. For MA-SZZ, Fan et al. [9] find that the labeled data has low false positive and false negative rates, compared to AG-SZZ, which contains a much larger number of false negatives. The low false positive and false negative rates may not be likely to impact the prediction of the MA models [9]. Checking the impact of performance bugs on the NFBugs data is an avenue for future work to advance studies pertaining non-functional bugs.

7.3 Studying non-functional bugs

Software quality research and practice concerns itself with a variety of different types of bugs. Non-functional bugs, including those of performance and security can be particularly costly bugs [18]. Tools can help reduce the cost overhead caused by these bugs [17]. In this study we concentrate on the applicability of the SZZ approach to determine the root cause of these non-functional bugs.

Jin et al. [17] studied real-world performance bugs to better guide software practitioners. Their findings show that developers need tool support to automatically fix such types of performance issues [17]. They also find that performance issues of newer software versions can be inherited easily from previous versions. This study calls for further and more detailed research on performance diagnosis, performance testing, and performance issue detection. Our study contributes to furthering software performance research by evaluating a tool to help developers automatically locate buggy code in the software. In theory, the SZZ approach should be able to locate at which point in time non-functional bugs, including performance issues, are introduced from previous versions, given that the performance issue has been detected.

Zaman et al. [18] conduct both qualitative and quantitative studies on performance and non-performance issues in Mozilla Firefox and Google Chrome, two open source web browsers. The study aims to understand the difference between performance issues and non-performance issues. Their findings show that developers spend more time fixing performance issues rather than non-performance issues [18]. This study advocates the importance of identifying root causes for performance issues and evaluating the impact of changes on performance issues, which are one of the many types of non-functional software issues. Our study differs as we only analyze non-functional bugs. However, in Zaman et al.'s [18] study, they only differentiate performance and non-performance bugs, where performance is a subset of non-functional software features. In our study, for each bug reported, their type (e.g., performance, security) has already been reported by the NFBugs dataset, therefore the true bug-inducing changes can also be linked to a non-functional bug type.

Nistor et al. [16] studied software performance since performance is critical for how users perceive the quality of software products. Performance bugs lead to poor user experience and low system throughput [36,37]. Their study includes how performance bugs are discovered, fixed, and compares the results with those for non-performance bugs from three popular code bases: Eclipse JDT, Eclipse SWT and Mozilla Firefox [38, 39].

Their results include suggestions of techniques to help developers reason about performance and suggest that better profiling techniques are needed for discovering performance bugs. Our study on the evaluation of an SZZ approach on non-functional bugs, can help determine whether it is reliable for developers to use references to past inducing code from past performance bugs to locate and fix new bugs with the help of an SZZ-implemented tool.

The absence of bug-inducing knowledge in issue trackers forces researchers to rely on alternative sources of information, such as the SZZ approach, which can be used as a heuristic approach to identify bug-inducing changes [24]. In a recent systematic literature review, it was determined that few researchers have made their SZZ implementations publicly available, causing extra research effort to be spent, as new projects based on SZZ output need to initially re-implement the approach [24]. The repeated re-implementation of SZZ also raises the risk that newly developed SZZ implementations have not been properly tested [24]. Borg et al. [24] present SZZ Unleashed, an open implementation of the SZZ approach for Git repositories. Our paper uses the same implementation of SZZ as McIntosh et al. [26].

Automatic identification of the differences between two versions of a file is a common and basic task in several applications of mining code repositories, commonly by using the git diff command [40]. Nugroho et al. [40] empirically analyze the impact of diff algorithms in three major applications: code churn metrics of

Table 6: Our key findings with comparison to previous literature.

Our Findings	Findings in Previous Literatures	Implications
Our findings show that among the 376 identified bug-inducing commits for 123 bugs, only 79 true positive bug-inducing commits are identified by the SZZ approach. Only 40 bugs have fully correctly identified bug-inducing changes.	Fan et al. [9] find that the percentage of the false positives in the bug-inducing changes labeled by MA-SZZ is 1%-6%. Borg et al. [24], state that JIT bug prediction using SZZ, corresponding to an F1 score of 0.10–0.15 is insufficiently accurate to be of practical value for developers. At the same time the classifier would miss too many truly bug-introducing commits.	While Fan et al. [9] find that the false positives percentage in the bug-inducing changes labeled by MA-SZZ is 1%-6%, we found a rate of 79%. Similarly to Borg et al. [24], we find that the false positives of the SZZ approach are too numerous for developers to trust the predictions. Furthermore, we also find that MA-SZZ missed many truly bug-inducing commits.
With MA-SZZ, we find a **67.5%** disagreement ratio for non-functional bugs, where a bug counts as a disagreement if all of the candidate bug-inducing changes identified by the SZZ approach for that bug are classified as incorrect.	For MA-SZZ, Costa et al. [10] report a **0%** to **17%** disagreement ratio.	The large difference in the disagreement ratio would be an important complement to previous literature.
We manually identify three reasons for falsely identified bug-inducing commits for non-functional bugs including 1) multi-purposes bug-fixing commits, 2) bug already being there, and 3) changes not related to the bug.	-	Although we find that these same reasons are not unique to falsely identified non-functional bugs, we show that the proportions of these reasons are expressed differently in functional bugs and non-functional bugs.
Non-functional bug-inducing commits differ from functional bugs inducing commits. Guidelines to identify falsely detected bug-inducing commits by the SZZ approach such as Earliest bug appearance and Realism of bug introduction cannot be reliably used when using SZZ to detect non-functional bugs.	Costa et al. [10] propose a framework to evaluate the implementations of the SZZ approach, comprised of three criteria: (1) earliest bug appearance, (2) future impact of changes, and (3) realism of bug introduction.	Our findings show that the prior guidelines **may not be useful** for non-functional bugs.
We find that 60 out of 82 of the true bug-inducing commits for the non-functional bugs in Java and 21 out of 32 in Python, are actually the initial version of the corresponding code snippet. In other words, the non-functional bugs were induced when developers first introduced the code into the project.	-	We find that this phenomenon is not found in functional bugs.

the SZZ approach, and patches extraction. The results of locating bug-inducing changes using the SZZ approaches relies on the diff results. Nugroho et al. [40] find that 25% of purposes of using the git diff command is for identifying bug-inducing change identification in the SZZ approach. Meanwhile, the work presented in this paper evaluates the detected-bug-inducing changes of the SZZ approach.

8 Threats to Validity

In this section we discuss the threats to the validity of our research.

External validity.

Threats to external validity are concerned with the extent to which we can generalize our results. Our dataset contains a total of 65 open source GitHub projects: 40 Java projects and 25 Python projects. It is possible that our results might not generalize to all programming languages. However, since our dataset consists of both Java and Python projects, we are confident that our results should have the potential to be generalized to projects of other languages. Furthermore, to further mitigate any language bias, the SZZ approach we used is language agnostic. While Ohira et al.'s [28] dataset can be seen as complementary to the NFBugs dataset, it does not contain enough information in terms of bug descriptions, for us to fully be sure of the results when we do a manual evaluation as some information (the cause of the bug) that is available in the NFBugs dataset is missing. We require this information since we are not domain experts for the projects in the dataset. It should be possible, however, for domain experts to use Ohira et al.'s [28] dataset to replicate our study. For future work, we plan to extend our study of non-functional bugs from other datasets, e.g., the data from Ohira et al. [28].

While the reasons for falsely detected bug-inducing commits for non-functional bugs presented in this paper are shown to have an effect on the detection of non-functional bugs, we do not claim that these reasons are unique to non-functional bugs. Because non-functional bugs are the focus of this paper, we did not study how these reasons could affect functional bugs.

Construct validity.

Threats to construct validity are concerned with the validity of our conclusions within the constraints of the dataset we used. The dataset used for this paper contains a total of 65 open source GitHub projects: 40 Java projects and 25 Python projects. Very few of the projects have an issue tracking system, and so for many, looking for the creation time of bug reports for a bug in the system was inapplicable. For these cases, we use the time when a bug fix was introduced instead of a bug report creation time to calculate the span between the introduction of a bug to the fix of that bug. A total of 27 of the bugs in the dataset had fully wrong identified bug-inducing changes, so we looked for false negatives, which are truly bug-inducing bugs that have been missed and then included in the augmented NFBugs dataset. We may still miss some information that introduced the non-functional bugs, which we attempt to mitigate by having each reviewer study the bug fix description and the nature of the bug in order to evaluate whether the identified changes from the SZZ approach induced the bug along with the methods being reported in the bug descriptions provided by the NFBugs dataset.

Internal validity.

Threats to internal validity are concerned with how our experiments were designed. Our manual analysis of the candidate bug-inducing commits for known bug fixing commits were subject to our own opinion and could therefore be biased by the opinion of the experimenter. In order to mitigate bias, we had three reviewers analyze the candidate bug-inducing commits separately and in parallel. Following our manual analysis, we compute the Multi Kappa Fleiss scores of the agreement of the three individuals. We obtained moderate to strong levels of agreement. The reviewers later met together to discuss disagreements. These measures allow us to mitigate and measure the internal bias of our manual study. Moreover, for the bugs where the SZZ approach was not able to identify any truly bug inducing commits, we tried our best manually to look in the repository to find the commits that were bug-inducing for bugs. It is still possible that we may miss some bug inducing commits.

Nugroho et al. reported that different diff algorithms produce different bug-fix commit identification and they show that the histogram diff is better than the default diff setting in Git. However, in this paper, we used the default diff setting in Git because we wanted to allow a comparison to prior studies [10] with as few confounding factors as possible. Future work and commercial approaches, should consider using histogram diff rather than the default diff setting to further improve the results presented in this work.

9 Conclusion

In this paper we compared our evaluation of the SZZ approach to prior work that evaluates the approach on functional bugs. We use the NFBugs dataset as a ground truth for non-functional bugs. The NFBugs dataset contains 65 open source GitHub projects: 40 Java projects and 25 Python projects. We examine the 89 listed Java bugs and 43 listed Python bugs to uncover root causes for false bug-inducing commit detection that have not been found by previous studies. Furthermore, we conduct an empirical study to evaluate the performance of the SZZ approach in terms of its ability to locate bug-inducing commits in the code on the 132 listed bugs. Finally, we manually look at the results and discuss their implications.

Our findings show that the vast majority (297 out of 376) of the automatically identified bug-inducing commits by the SZZ approach for non-functional bugs are false positives. In addition, although there exists guidelines from prior study to assist in automatically identify falsely detected bug-inducing commits for functional bugs, these guidelines cannot be reliably used for non-functional bugs. Finally, the existing improvements to SZZ approach cannot help improve identifying the bug-inducing commits for non-functional bugs.

Our paper is the first to focus exclusively on the use of the SZZ approach on non-functional bugs. Moreover, we augment the NFBugs dataset by adding a field to each bug description introducing bug-inducing commits that we manually analyzed as truly bug-inducing. By extending the dataset, we hope this information proves useful to help future research in locating bug-inducing commits, particularly with respect to non-functional bugs. Our findings indicate that new or adjusted tooling should be designed by considering the unique characteristics of non-functional bugs in order to accurately identify their bug inducing changes.

References

1. T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, Oct 2005.
2. A. E. Hassan, “Predicting faults using the complexity of code changes,” in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 78–88.
3. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.
4. T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: A study of developer work habits,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 492–501.
5. P. Grubb and A. Takang, *Software maintenance - concepts and practice (2. ed.).*, 01 2003.
6. S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, “Automatic identification of bug-introducing changes,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, Sep. 2006, pp. 81–90.
7. J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.
8. M. Kim and E. Lee, “Are information retrieval-based bug localization techniques trustworthy?” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 248–249. [Online]. Available: https://doi.org/10.1145/3183440.3194954
9. Y. Fan, X. Xia, D. Alencar da Costa, D. Lo, A. E. Hassan, and S. Li, “The impact of changes mislabeled by szz on just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
10. D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, “A framework for evaluating the results of the szz approach for identifying bug-introducing changes,” *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, July 2017.
11. E. Neto, D. Costa, and U. Kulesza, “The impact of refactoring changes on the szz algorithm: An empirical study,” 03 2018.
12. M. Hamill and K. Goseva-Popstojanova, “Exploring the missing link: An empirical study of software fixes,” *Softw. Test. Verif. Reliab.*, vol. 24, no. 8, p. 684–705, Sep. 2014. [Online]. Available: https://doi.org/10.1002/stvr.1518
13. L. Williams, G. McGraw, and S. Migues, “Engineering security vulnerability prevention, detection, and response,” *IEEE Software*, vol. 35, no. 5, pp. 76–80, Sep. 2018.
14. H. Mahrous and B. Malhotra, “Managing publicly known security vulnerabilities in software systems,” in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, Aug 2018, pp. 1–10.
15. Liu Ping, Su Jin, and Yang Xinfeng, “Research on software security vulnerability detection technology,” in *Proceedings of 2011 International Conference on Computer Science and Network Technology*, vol. 3, Dec 2011, pp. 1873–1876.
16. A. Nistor, T. Jiang, and L. Tan, “Discovering, reporting, and fixing performance bugs,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 237–246.
17. G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, “Understanding and detecting real-world performance bugs,” *SIGPLAN Not.*, vol. 47, no. 6, pp. 77–88, Jun. 2012.
18. S. Zaman, B. Adams, and A. E. Hassan, “Security versus performance bugs: A case study on firefox,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. New York, NY, USA: ACM, 2011, pp. 93–102.
19. “What is non functional testing? types with example.” [Online]. Available: https://www.guru99.com/non-functional-testing.html
20. A. Radu and S. Nadi, “A dataset of non-functional bugs,” in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR ’19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 399–403.
21. M. Glinz, “On non-functional requirements,” in *15th IEEE International Requirements Engineering Conference (RE 2007)*, Oct 2007, pp. 21–26.
22. S. Kim and E. J. Whitehead, Jr., “How long did it take to fix bugs?” in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR ’06. New York, NY, USA: ACM, 2006, pp. 173–174.

23. K. Pan, S. Kim, and E. J. Whitehead, Jr., “Toward an understanding of bug fix patterns,” *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009.
24. M. Borg, O. Svensson, K. Berg, and D. Hansson, “Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project,” in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, ser. MaLTeSQuE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 7–12. [Online]. Available: https://doi.org/10.1145/3340482.3342742
25. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, june 2013.
26. S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018.
27. C. Rosen, B. Grawi, and E. Shihab, “Commit guru: Analytics and risk prediction of software commits,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 966–969. [Online]. Available: https://doi.org/10.1145/2786805.2803183
28. M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, “A dataset of high impact bugs: Manually-classified issue reports,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 518–521.
29. S. Davies, M. Roper, and M. Wood, “Comparing text-based and dependence-based approaches for determining the origins of bugs,” *Journal of Software: Evolution and Process*, vol. 26, 01 2014.
30. G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*, 1st ed. Wiley Publishing, 1998.
31. M. McHugh, “Interrater reliability: The kappa statistic,” *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, vol. 22, pp. 276–82, 10 2012.
32. C. Williams and J. Spacco, “Szz revisited: Verifying when changes induce fixes,” in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, ser. DEFECTS ’08. New York, NY, USA: ACM, 2008, pp. 32–36.
33. D. Steidl, B. Hummel, and E. Juergens, “Incremental origin analysis of source code files,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 42–51. [Online]. Available: https://doi.org/10.1145/2597073.2597111
34. C. Neto and E. Barbalho, “Enhancing the szz algorithm to deal with refactoring changes,” 2018.
35. Jpace, “jpace/diffj.” [Online]. Available: https://github.com/jpace/diffj
36. I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*, 1st ed. O’Reilly Media, Inc., 2009.
37. R. E. Bryant and D. R. O’Hallaron, *Computer Systems: A Programmer’s Perspective*, 3rd ed. Pearson, 2015.
38. J. team, “Eclipse java development tools (jdt).” [Online]. Available: https://www.eclipse.org/jdt/
39. C. Guindon, “Swt: The standard widget toolkit.” [Online]. Available: https://www.eclipse.org/swt/
40. Y. S. Nugroho, H. Hata, and K. Matsumoto, “How different are different diff algorithms in git?” *Empirical Software Engineering*, vol. 25, no. 1, p. 790–823, Sep 2019. [Online]. Available: http://dx.doi.org/10.1007/s10664-019-09772-z