Benefits and pitfalls of token-level SZZ: An empirical study on OSS projects

Hiroya Watanabe*, Masanari Kondo†, Eunjong Choi* and Osamu Mizuno*
*Kyoto Institute of Technology, Kyoto, Japan
†Kyushu University, Fukuoka, Japan
Email: h-watanabe@se.is.kit.ac.jp, kondo@ait.kyushu-u.ac.jp, echoi@kit.ac.jp, o-mizuno@kit.ac.jp

***Abstract*—SZZ is the de facto standard method for identifying bug-inducing commits. The accuracy of this method heavily relies on source code management systems, such as Git, as it requires tracing the history of source code changes (i.e., commit histories) to bug-inducing commits. However, it has been reported that these systems introduce biases in commit histories because they only store line-level changes. It is known that such coarse-grained line-level changes can result in the failure to accurately track the commit history and reduce the performance of SZZ. To relieve this challenge, we explore the accuracy of SZZ in token-level changes, which provide finer-grained information to trace commit histories compared to line-level ones, and we discuss the potential benefits and pitfalls of utilizing token-level changes for SZZ. As a result of experiments on 68 OSS projects, we found that SZZ, which uses token-level histories, identifies two new bug-inducing commits that are missed when using line-level histories. Furthermore, our manual analysis of the identified commits indicates that they reduce false-positive bug-inducing commits caused by source code formatting and whitespace changes. However, this improvement in detecting bug-inducing commits comes with a trade-off of 0.081 decrease in overall accuracy, as measured by the F1 score. Consequently, we summarized three potential benefits and five pitfalls of using token-level and line-level tracking for SZZ.**

***Index Terms*—SZZ, bug inducing commit, software repository mining, empirical study**

I. Introduction

Bugs are unintentionally introduced into software systems despite the efforts of software developers to prevent them. These bugs result in unintended software behavior [1]–[3] and have significant financial consequences for software development companies [4], [5]. As a result, bugs have been a major area of research in software engineering. Previous studies [6]–[10] have aimed to assist software developers by analyzing, evaluating, and predicting bugs in software systems in order to minimize the occurrence of bugs.

Bug datasets are essential for promoting such studies, and there are numerous bug datasets available. One example is the Defects4J dataset [11], which contains bugs gathered from real-world Java systems. Although these datasets ensure high quality by collecting bugs from real-world scenarios, their sizes are relatively small due to the cost of data collection, which limits potential research extensions. Consequently, the *SZZ* method [12] is commonly employed to construct large-scale bug datasets. It automatically identifies changes in source code management (SCM) systems that introduce bugs (*bug-inducing changes*). Furthermore, these changes enable the detection of source code revisions that contain bugs.

While SZZ can increase the size of bug datasets, it also reduces the quality of the dataset. This is because the accuracy of SZZ heavily relies on the change histories in SCM systems, which can sometimes be misleading. SCM systems only store line-level changes, which means information about changes within a line can be lost [13]–[15]. SZZ tracks bug-inducing commits based on these change histories. However, misleading histories can easily disrupt these tracks, resulting in false-positive or false-negative bug-inducing changes. To address this challenge, it is crucial to improve the accuracy of tracking change histories.

In this paper, we utilize token-level change histories to enhance the accuracy of tracking change histories for SZZ (*token-level tracking*). It is worth noting that token-level change histories have the potential to improve the accuracy of tracking compared to tracking on traditional line-level change histories (*line-level tracking*) [13]. More specifically, we transform line-level change histories into token-level change histories using cregit, a tool developed in a previous study [13]. We empirically compare the impact of token-level tracking and line-level tracking on the accuracy of SZZ in 68 open-source software (OSS) projects based on the following research questions (RQs).

- RQ1 To what extent do token-level changes improve the overall accuracy of SZZ?
- RQ2 What are the reasons for the differences in the true positive commits identified using line-level and token-level tracking?
- RQ3 What are the reasons for the differences in the false positive commits identified using line-level and token-level tracking?

Based on our experiments, we discovered that using token-level histories helps identify two new bug-inducing commits that are missed when using line-level histories. However, this improvement in detecting bug-inducing commits comes at the cost of 0.081 decrease in overall accuracy, as measured by the F1 score.

To understand the reasons behind these differences, we conducted an in-depth analysis. We manually inspected the true-positive and false-positive commits identified by both token-level and line-level tracking. Our findings indicate that token-level tracking can effectively reduce false-positive bug-

TABLE I: Overview of SZZ variants in previous studies

Name	Year	Key characteristics
AG-SZZ [16], [17]	2006	Exclude formatting and appearance changes
DJ-SZZ [18]	2008	Ignore non-executed line changes such as whitespace
R-SZZ [19]	2014	Return the most recent commit
L-SZZ [19]	2014	Return the commit with the highest number of changed lines
MA-SZZ [3]	2017	Exclude meta changes (e.g., branch changes)
RA-SZZ [20], [21]	2019	Exclude refactoring changes
PR-SZZ [22]	2022	Use the pull request information
Rosa et al. [23]	2023	A heuristic to handle lines in bug-fixing commits
Neural SZZ [24]	2023	Use only highly important deleted lines

inducing commits caused by source code formatting and whitespace changes, which are identified by line-level tracking. However, token-level tracking may overlook bug-inducing commits that were able to be modified by adding tokens.

The contributions of this paper are as follows.

1) This paper is the first study to compare the accuracy of the SZZ method using either line-level or token-level tracking.
2) We empirically explore the potential benefits and pitfalls of using line-level and token-level tracking for SZZ. Consequently, we summarize three potential benefits and five pitfalls. This knowledge will support future studies to enhance SZZ.

The organization of our paper is as follows. Section 2 introduces related works and provides a description of the motivating example. Section 3 describes the dataset used in our study. Section 4, 5, and 6 present the motivations, approach, and results for RQ1, 2, and 3. Section 7 provides a summary of the lessons learned from our study. Section 8 describes threats to validity. Section 9 presents the conclusion.

II. Background

A. Related Work

SZZ methods. The SZZ method [12] is widely used for identifying bug-inducing commits. The SZZ method involves two steps: (1) identifying bug-fixing commits and (2) identifying bug-inducing commits. In the first step, bug reports are linked to commits, and these linked commits are considered as bug-fixing commits corresponding to the bug reports. The second step involves tracking back to bug-inducing commits from the bug-fixing commits in the commit history of the SCM system. If Git is used as the SCM system, the *blame* command is used to trace the commit history. However, it still has some drawbacks that need to be addressed. For example, the blame command may not correctly track lines that have been changed multiple times [13] and may identify false-positive bug-inducing commits.

Hence, previous studies have proposed various SZZ variants to alleviate its drawbacks [3], [16]–[23]. Table I provides an overview of the previous studies. Kim et al. [16] proposed AG-SZZ, which uses annotation graphs [17] to exclude formatting and appearance changes that are irrelevant to the actual modification. Williams et al. [18] proposed DJ-SZZ, which ignores non-executed line changes such as whitespace, comments, and changes in import statements using mapping of line numbers and the Java syntax-aware diff tool, DiffJ [25]. Neto et al. [20], [21] proposed RA-SZZ, which excludes refactoring changes using the refactoring detection tools Refdiff [26] and RefactoringMiner [27]. Tang et al. [24] proposed Neural SZZ, which ranks the deleted lines of bug-fixing commits based on their importance. It then uses only highly important lines as input to SZZ. However, there is still room for improvement. For example, previous studies use line-level changes stored in the SCM system, which potentially overlooks changes within a line [13]–[15], [28].

Datasets. Although various SZZ variants have been proposed, the quality of the ground truth data is a challenge to evaluate these variants [3]. Hence, previous studies have evaluated the impact of ground truth data quality on SZZ [23], [29], [30]. Herzig et al. [30] reported that 33.8% of all bug reports do not actually refer to bugs, but rather serve other purposes, such as proposing a new feature. Furthermore, they found that this mislabeling leads to a 39% rate of false-positive bug-inducing files. Tantithamthavorn et al. [29] investigated the impact of mislabeling in the dataset generated by SZZ on the performance of defect prediction models. They discovered that models trained on the mislabeled dataset achieve only 56-68% of the recall of the models trained on a clean dataset. Rosa et al. [23] have created a high-quality dataset for evaluating the SZZ method, focusing only on information provided by the developer. Additionally, they have assessed two new heuristics for SZZ using this dataset. Due to the significance of the ground truth data, in this study, we utilized a high-quality dataset provided by Rosa et al. [23] and manually validated the identified commits.

B. Motivating Example

Listing 1 demonstrates the example that served as the motivation for this study. The bug-fixing commit (Listing 1a) corrects the 7th argument of the `printDependency` function from `null` to `classifier`[1]. The `null` value induces a bug. However, the traditional SZZ method fails to identify the commit that introduced this bug. The identified bug-inducing commit by the traditional SZZ method (Listing 1b) adds `dds[i].isTransitive()` to the 9th argument of the `printDependency` function[2]. As the 7th argument of the `printDependency` function is already `null` in this commit, it indicates that a bug has already been introduced.

The main reason for this false-positive bug-inducing commit is that the SZZ method tracks the commit history at the line level, even though the cause of the bug is a token. As explained in Section II-A, the SZZ method looks for bug-inducing commits by tracing back from bug-fixing commits in

[1]https://github.com/apache/ant-ivy/commit/1f0c99d0e012d84863e6a818facb143c9f03fac3

[2]https://github.com/apache/ant-ivy/commit/e484646c60eaca1f89921db7058d8927302d7226

```
264 + final String classifier =
    ↪  dds[i].getExtraAttribute("classifier");
265   printDependency(out, indent,
    ↪  mrid.getOrganisation(), mrid.getName(),
265 -     mrid.getRevision(), null, null, scope,
    ↪  optional, dds[i].isTransitive(),
266 +     mrid.getRevision(), null, classifier,
    ↪  scope, optional, dds[i].isTransitive(),
```

(a) Bug-fixing commit

```
253   printDependency(out, indent,
    ↪  mrid.getOrganisation(), mrid.getName(),
254 -     mrid.getRevision(), null, null, scope,
    ↪  optional, excludes);
265 +     mrid.getRevision(), null, null, scope,
    ↪  optional, dds[i].isTransitive(),
266 +     excludes);
```

(b) False-positive bug-inducing commit

```
259 + printDependency(out, indent,
    ↪  mrid.getOrganisation(), mrid.getName(),
260 +     mrid.getRevision(), null, null, scope,
    ↪  optional, excludes);
```

(c) Bug-inducing commit

Listing 1: Motivating Example

the commit history of the SCM system, which usually records line-level changes.

Figure 1 illustrates a scenario where the git blame command, currently the most popular method for tracing commit history in SZZ, fails to identify bug-inducing commits. The red token represents the updated token for each commit. `buggy_arg` is the cause of the bug, and `clean_arg` is the modified token. Commit C introduces a buggy argument on line 10 (i.e., bug-inducing commit), and Commit B updates another argument. In this scenario, when using the git blame command on the bug-fixing commit Commit A, Commit B is identified as the bug-inducing commit because it updates line 10. If the blame command could track the commit history at the token level, as shown in Figure 1, the SZZ method would successfully identify the bug-inducing commit Commit C because `buggy_arg` is not updated by Commit B.

Indeed, token-level tracking can successfully identify the bug-inducing commit in the motivating example (Listing 1c)[3]. This finding suggests that using finer-grained tracking, specifically at the token-level, may enhance the performance of the SZZ method. However, previous studies have overlooked this fact. Therefore, our study aims to investigate the potential benefits and pitfalls associated with token-level tracking in the SZZ method.

[3]https://github.com/apache/ant-ivy/commit/9c7f6d421223e6f76e67729b01ecc7f356eb3a29

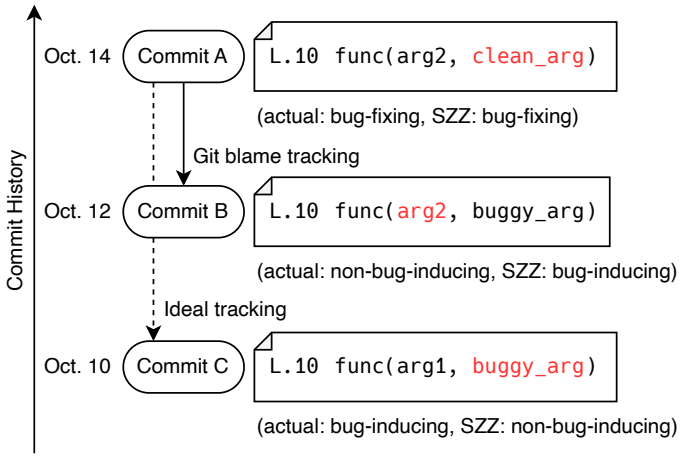


Fig. 1: Different scenarios in which SZZ identifies correct and incorrect bug-inducing commits

III. Dataset Preparation

In this section, we introduce the studied dataset. We first describe the ground-truth data, which includes commits that were annotated as either bug-inducing or not. Then, we introduce cregit, a tool for implementing the token-level SZZ method. Finally, we describe the commits that were automatically annotated using the token-level and line-level SZZ methods.

A. Ground-Truth Data

We utilize the *developer-informed oracle dataset* [23] as the ground truth for pairs of bug-fixing commits and bug-inducing commits. This dataset was created with the goal of providing more accurate pairs of bug-fixing commits and bug-inducing commits by focusing on information provided by developers. Specifically, it collects bug-fixing commits whose commit messages include the commit hash of the commit where the bug was introduced. We selected this dataset to relieve the concern of the bug data quality [3], [23].

The overall developer-informed oracle dataset covers eight main programming languages and a total of 1,854 repositories. In this experiment, we specifically selected 72 repositories that meet the criterion of being mainly developed in Java.

The reason for filtering the projects is to conduct manual analyses on the commits identified by the token-level tracking. It is challenging to automatically assess the accuracy of tracking because we need to interpret commits to determine if the identified commits are the correct bug-inducing commits associated with the bug-fixing commits. To make the manual analyses feasible, we filter the projects first. For this filtering, we select the criterion of being mainly developed in Java. This is because previous studies have also utilized datasets consisting of projects developed in Java to assess their SZZ variants [3], [16]–[18], [20], [21]. Additionally, many open-source SZZ implementations specifically focus on Java [31]–[33].

From the selected 72 projects, we exclude four projects for the following reasons.

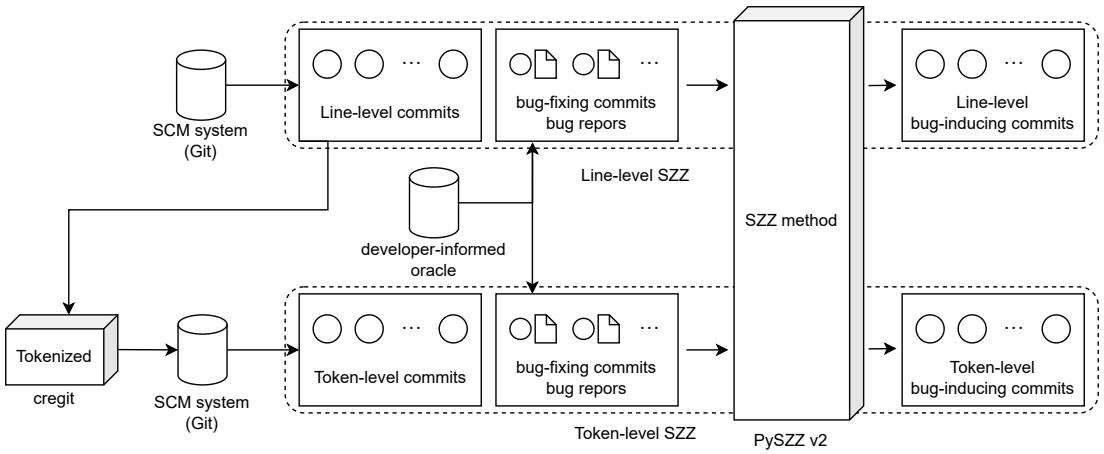


Fig. 2: Overview of the annotation using the line-level and token-level SZZ methods

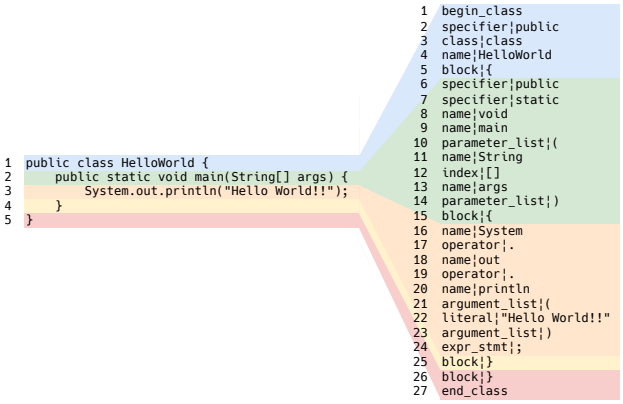


Fig. 3: Example of tokenization by cregit (the target code displays "Hello World!!" in Java)

- The repository does not exist.
  - Kasukoi/eisverkauf
- The bug-fixing commit was removed.
  - imagej/imagej
- The cregit process entered an infinite loop and did not terminate.
  - JetBrains/intellij-community
  - brianchandotcom/liferay-portal

Finally, our ground truth dataset consists of 68 repositories with 997,287 commits, including 76 bug-inducing ones.

B. cregit

The SZZ method for Git repositories uses the blame command to track the commit history, allowing retrieval of various data about the latest previous changes of a given line (such as commit hashes, authors, and dates). However, the blame command is applicable only to entire lines. This level of granularity overlooks changes for each line, such as individual tokens [13]–[15]. To address this issue, German et al. developed cregit, a tool that tracks changes at the token level [13]. While tracking changes at the line level identifies the commits that introduced the tokens with an accuracy ranging from 75% to 91%, token-level tracking achieves a higher accuracy of 94.5% to 99.2%.

cregit parses the source code file and generates an output file that contains all the original tokens. Each line in the file represents a token. This operation is performed on all source code files in every commit, creating new *view commits* that include the outputted files along with the original metadata, such as the original commit message, creation date, and author information. Each original commit corresponds to a converted view commit on a one-to-one basis. Figure 3 shows an example of source code tokenized by cregit. The code in the example displays "Hello World!!" in Java.

Except for unique tokens of cregit such as `begin_class` and `end_class`, which indicate the beginning and end of a class, a single line consists of the token type and the corresponding token, separated by the "|" symbol. For example, in the second tokenized line, `specifier|public`, `specifier` is the type and `public` is the corresponding token. The order of each line is consistent with the order of the tokens as they appear in the original source code file.

The advantage of using cregit in this analysis is that we can directly apply existing SZZ methods to the view commits. cregit converts source code files and generates view commits, which preserve all the information of the original commit except for the original source code files. As a result, we can utilize all git commands, including the blame command, on the view commits. When comparing line-level and token-level, all we need to do is replace the target repository from the original repository with the repository that includes view commits. Because of this advantage, we use cregit to compare line-level and token-level SZZ methods. It is worth noting that cregit has already been used in mining software repositories [34]–[36].

C. Bug-inducing commits annotated by the SZZ method

Repositories. To execute the SZZ method, we collect repositories to construct our studied dataset. We clone 68 repos-

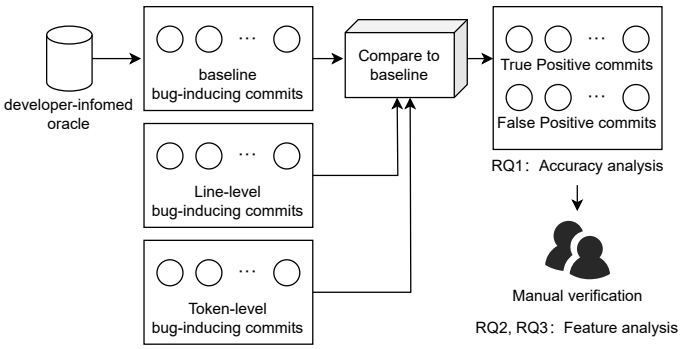


Fig. 4: Overview of our research methodology

itories and check out the commit that was used to create the developer-informed oracle dataset. This ensures that the dataset avoids including bug-inducing and bug-fixing commits that are not included in the developer-informed oracle dataset. During this process, we discovered a repository that lacks the commit used to create the developer-informed oracle dataset. To address this issue, we choose the closest commit to the missing one. The following is information about the repository and the new commit.

- UniTime/unitime
  - Hash : af1398224289l6fb7c0efc804dacbc7c9dcdfbc1
  - Commit Date : Aug 3 17:56:14 2022

Figure 2 presents an overview of the annotation using the line-level and token-level SZZ methods. We describe the details of these methods below.

Line-level SZZ. We utilized PySZZ v2[4] as the tool for the base SZZ method. PySZZ v2 is a Python-based tool developed by the researchers who created the developer-informed oracle. It includes input/output utilities for handling data from the developer-informed oracle, making it user-friendly. PySZZ v2 also provides various implementations of SZZ methods, which can be switched through configuration files. For this study, we used the basic SZZ method and applied a filter that utilizes bug report dates (i.e., `bszz_issue_filter.yml`) as our line-level SZZ method.

Token-level SZZ. To implement the token-level SZZ method, we use cregit to extract tokens from the source code and convert the history of the Git repository from line-level changes to token-level changes. We then apply PySZZ v2 to these tokenized repositories. As PySZZ v2 operates on tokenized repositories, we consider this our token-based SZZ method.

IV. RQ1: To what extent do token-level changes improve the overall accuracy of SZZ?

A. Motivation and Approach

While various previous studies have aimed to enhance the SZZ method, there has been no research investigating how performance is affected by token-level tracking. To gain an initial understanding of the impact of token-level tracking on SZZ performance, we compare the accuracy of identifying bug-inducing commits between line-level and token-level SZZ methods in this RQ. We assess accuracy in terms of precision, recall, and F1-score, considering true-positive and false-positive bug-inducing commits. The ground truth data is the developer-informed oracle dataset.

Furthermore, we investigate not only the overall performance but also the overlap of their identified commits. We thoroughly examine the unique and intersected true-positive and false-positive commits.

Finally, we use McNemar's test to analyze the statistical significance of the differences between correctly identified commits using line-level and token-level SZZ methods. McNemar's chi-square statistic can be calculated using the following formula:

$$\chi^2 = \frac{(b-c)^2}{b+c} \tag{1}$$

where b indicates the number of false-negative commits of line-level, while c indicates the number of true-positive commits of token-level. The results are considered statistically significant at $p < 0.05$.

Figure 4 presents an overview of our research methodology.

B. Result

Performance. Table II shows a 0.065, 0.105, and 0.081 reduction in precision, recall, and F1 score, respectively. While our motivation example demonstrates the potential of the token-level SZZ method, the overall performance indicates that the line-level SZZ method outperforms the token-level SZZ method. Additionally, the number of true positives (TP) decreases while the number of false positives (FP) increases.

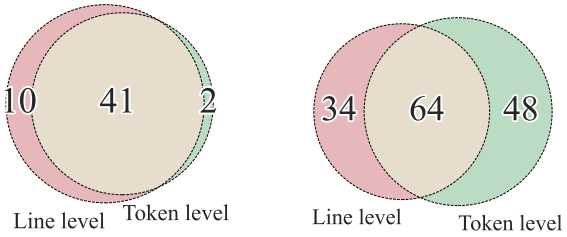
Overlap. The overlaps of the identified bug-inducing commits are illustrated in Figure 5. The red and green areas indicate the proportion of commits that are only identified by either the line-level or token-level method. The center area represents the overlap. Figure 5a indicates that a majority of the identified bug-inducing commits are intersected between both methods, but there are some differences. While the line-level SZZ method identifies 10 unique true-positive bug-inducing commits, the token-level SZZ method identifies 2 unique true-positive bug-inducing commits as well. We suspect that although the token-level SZZ method may have an overall worse performance compared to the line-level method, it can still identify commits that are not effectively identified by the line-level method. However, the token-level SZZ method discovers 14 additional false-positive commits compared to the line-level SZZ method (Figure 5b). Therefore, using token-level tracking may result in more false-positive identifications of bug-inducing commits from bug-fixing commits.

McNemar's test. The results of McNemar's test are as follows: $\chi^2 = 4.765$, and $p = 0.029$. Since the p-value is less than 0.05, the identified commits between line-level and token-level are different.

[4]https://github.com/grosa1/pyszz_v2

TABLE II: Comparison of the accuracy of identified bug-inducing commits

	Precision	Recall	F1 score	TP	FP
Line level	0.342	0.671	0.453	51	98
Token level	0.277	0.566	0.372	43	112
Difference	-0.065	-0.105	-0.081	-8	+14



(a) True Positive (b) False Positive

Fig. 5: Overlap of identified bug-inducing commits

Compared to the line-level SZZ method, unfortunately, the token-level SZZ method results in worse overall performance, with a 0.081 decrease in F1 score. On the other hand, the token-level method can discover additional true-positive commits that may not be identified by the line-level SZZ method. As described in Section II-B, this indicates that token-level tracking has the potential to enhance the SZZ performance.

V. RQ2: What are the reasons for the differences in the true positive commits identified using line-level and token-level tracking?

A. Motivation and Approach

RQ1 shows that the token-level SZZ has the potential to enhance the SZZ performance since it can discover unique true-positive bug-inducing commits. While this may support the potential benefits as described in Section II-B, we do not manually verify the identified commits. In RQ2, we investigate the characteristics of these unique true-positive bug-inducing commits and compare them with the results obtained from the line-level SZZ method. Specifically, we examine why these true-positive bug-inducing commits are identified using either line-level or token-level tracking. The first two authors adopted an open coding process [37] to manually investigate the commits and summarize their characteristics to determine the reasons behind their identification.

B. Result

Line-level. We provide the reasons for the 10 true positive commits that were identified at the line-level but not at the token-level. The numbers in parentheses indicate the count of commits classified under each reason. All commits and their reasons are contained in the replication package[5].

[5]https://zenodo.org/doi/10.5281/zenodo.10044842

```
1928   if (Keyboard.getEventKey() ==
     ↪  Keyboard.KEY_ESCAPE) {
1929 -     if (!packetHandler.isLoadingLevel){
1929 +     if (!packetHandler.isLoadingLevel
     ↪  || !isOnline() ){
1930          pause();
```

(a) Changes at line-level

```
11405   name|packetHandler
11406   operator|.
11407   name|isLoadingLevel
11408 + operator|||
11409 + operator|!
11410 + name|isOnline
11411 + argument_list|()
11412   condition|)
11413   block|{
```

(b) Changes at token-level

Listing 2: Example of changes in a line-level and token-level bug-fixing commit in andrewphorn/ClassiCube-Client

- *Bug-fixing commits that include additions and removals at the line-level, but only include additions at the token-level* (8)
- *Bug-inducing commits that can be identified by tracking cosmetic changes, such as spaces and indentations* (2)

The most common reason is that bug-fixing commits only involve additions at the token-level. As an example, Listing 2 illustrates a bug-fixing commit from the andrewphorn/ClassiCube-Client repository[6]. The bug-fixing commit tracked by line-level is shown in Listing 2a, while the tokenized one is shown in Listing 2b.

This bug-fixing commit adds a condition `|| !isOnline()` to the if statement on line 1929 (Listing 2a). This modification at the line-level corresponds to the additions from line 11408 to 11411 at the token-level (Listing 2b). While this bug-fixing commit only adds a condition and the token-level tracking shows additions of four tokens, it is important to note that the line-level tracking includes one line deletion and one line addition. In this case, the token-level SZZ method is unable to track the bug-inducing commit from the bug-fixing commit using the blame command. This limitation arises from the fact that the blame command can only track lines that have been deleted. In other words, the blame command cannot track changes like the token-level commit in this case, which only involves additions without any deletions.

This is not a specific challenge in the token-level SZZ method, but a common challenge for all SZZ methods [23]. This limitation has been studied by Rezk et al. as *ghost commits* [28]. In particular, they refer to changes consisting only of deletion as *Mapping Ghost type 2 (MG2)*. In fact, there are bug-fixing commits that involve only adding lines. Therefore, this result suggests that even with a finer granularity, which we

[6]https://github.com/andrewphorn/ClassiCube-Client/commit/638ece305bdbd783e4c326011dd1446c737b536f

```
132 + logger.info("[eXist Home : "
133 +         + System.getProperty("exist.home", "unknown") + "]");
134  logger.info("[eXist Version : "
135           + SystemProperties.getInstance().getSystemProperty("product-version", "unknown") + "]");
136  logger.info("[eXist Build : "
137           + SystemProperties.getInstance().getSystemProperty("product-build", "unknown") + "]");
137 - logger.info("[eXist Home : "
138 -         + SystemProperties.getInstance().getSystemProperty("exist.home", "unknown") + "]");
138  logger.info("[Git commmit : "
139           + SystemProperties.getInstance().getSystemProperty("git-commit", "unknown") + "]");
```

(a) Changes at line-level tracking

```
843   operator|.
844   name|info
845   argument_list|(
846 - literal|"[eXist Version : "
846 + literal|"[eXist Home : "
847   operator|+
848 - name|SystemProperties
849 - operator|.
850 - name|getInstance
851 - argument_list|()
848 + name|System
849   operator|.
853 - name| getSystemProperty
850 + name| getProperty
851   argument_list|(
855 - literal|" product-version "
852 + literal|" exist.home "
853   argument_list|,
854   literal|"unknown"
855   argument_list|)
    @@ -864,7 +861,7 @@
861   operator|.
862   name|info
863   argument_list|(
867 - literal|"[eXist Build : "
864 + literal|"[eXist Version : "
865   operator|+
866   name|SystemProperties
867   operator|.
    @@ -873,7 +870,7 @@
870   operator|.
871   name|getSystemProperty
872   argument_list|(
876 - literal|"product- build "
873 + literal|"product- version "
```

(b) Changes at token-level tracking

Listing 3: Example of changes between bug-fixing commits in adamretter/exist repository

initially believed would allow for more detailed and accurate tracking, we may still encounter the same challenge observed in line-level tracking. The simplest approach to address this challenge is a hybrid tracking approach. For instance, in cases where the commit after tokenization consists only of added tokens, switching back the original line-based commits could be an effective method.

TABLE III: Comparison of the accuracy of identified bug-inducing commits when using the histogram algorithm

	Precision	Recall	F1 score	TP	FP
Line level	0.342	0.671	0.453	51	98
Token level	0.268	0.553	0.361	42	115
Difference	-0.074	-0.118	-0.092	-9	+17

Token-level. Next, we provide the reasons for the two true positive commits that were identified at the token-level but not at the line-level.

- *Accurate diffs in bug-fixing commits* (2)

As an example, we present one of the bug-fixing commits from the adamretter/exist project in Listing 3[7]. The line-level bug-fixing commit is shown in Listing 3a, and the tokenized commit is shown in Listing 3b. Due to the length of the diff in the token-level commit, only a portion of its distinctive part is included here.

This commit addresses an issue with the incorrect retrieval of the `exist.home` property by changing the order of logging output. The line-level commit recognizes this modification as a relocation of two lines from lines 137 and 138 to lines 132 and 133. In contrast, the token-level commit interprets this modification as a change in some tokens. This suggests that the token-level commit does not interpret this modification as a movement of lines. This diff difference results in the two true positive commits by the token-level SZZ method.

This result implies that the performance of the token-level SZZ method may depend on the diff algorithms used by Git. In fact, Git provides multiple algorithms to generate the diffs between two commits. For this experiment, we used the default algorithm: the mayer algorithm. However, using different diff algorithms can lead to different diffs [38] and different bug-inducing commits. Therefore, we thoroughly examine the differences between the diff algorithms provided by Git. A previous study [38] suggests using the histogram algorithm. In fact, applying the histogram algorithm changes the commit in Listing 3b to the commit where a single logger is moved similar to Listing 3a. Hence, we conduct the same experiment as RQ1 but with the histogram algorithm.

The results are presented in Table III. When comparing the results with the mayer algorithm, as shown in Table II, we

[7]https://github.com/adamretter/exist/commit/78d23a3263376611d7b065944fd7a5597bec025c

TABLE IV: The number of commits confirmed as true-positive by our manual verification within the false-positive by the SZZ method and the developer-informed oracle

Type	#commits confirmed as TP by manual verification/ #FP commits by the SZZ method
Line-level	0/34
Token-level	1/48
Both	3/64

observed no change in the results at the line-level. However, at the token-level, there was a decrease of one true positive commit and an increase of three false positive commits, resulting in a decrease in all evaluation measures. These results suggest that the choice of different diff algorithms has an impact, particularly when tracking at the token-level. Therefore, the performance of the token-level SZZ method could be improved by selecting a more suitable diff algorithm.

The token-level SZZ method does not address all the challenges that the line-level SZZ method faces. In particular, the token-level SZZ method still considers bug-fixing commits that only involve additions as potential threats. However, our experiment shows that the token-level SZZ method can identify two unique true-positive commits due to its more accurate diffs. Since the choice of diff algorithms can impact this accuracy, further research is needed to identify a suitable diff algorithm that can enhance the performance of the token-level SZZ method.

VI. RQ3: What are the reasons for the differences in the false positive commits identified using line-level and token-level tracking?

A. Motivation and Approach

In contrast to RQ2, this RQ focuses on analyzing false-positive bug-inducing commits. Similar to RQ2, the first two authors manually investigate the commits to gain insights into the differences between the line-level and token-level SZZ methods in terms of false-positive commits.

Before investigating the difference, we manually inspect and remove commits that are identified as false-positive by the SZZ method and the ground truth dataset (the developer-informed oracle), but are confirmed as true-positive through our manual verification. This is necessary because our ground truth dataset may incorrectly label actual bug-inducing commits as non-bug commits. To improve the validity of this RQ, we exclude these commits from the false-positive commits by the SZZ method.

Table IV presents the number of commits confirmed as true-positive by our manual verification within the false-positive by the SZZ method. Among them, we identified one commit in the false-positive commits induced by the token-level SZZ method only, none in those induced by the line-level SZZ method only, and three commits in those induced by both the token-level and line-level SZZ methods.

TABLE V: Percentage of commits that are accurately tracked but false-positive by the SZZ method

Type	#commits tracked accurately/ #FP commits by the SZZ method	Percentage
Line level	9/34	0.265
Token level	19/48	0.396
Both	44/64	0.688

Next, we manually investigate whether the token-level and line-level SZZ methods accurately track the change histories, even if the identified bug-inducing commits turn out to be false positives. This is because, in fact, there are commits that are not theoretically identified by the SZZ method. For example, the following *missing co-changes* scenario exists.

- While a function is modified, developers overlook to update the invocation.
- While a code fragment is modified, there are code clones that should also be updated.

The token-level SZZ method may generate false positive commits in the above scenario. Consequently, even if the token-level SZZ method accurately tracks commit histories, it can potentially reduce overall performance. If the token-level SZZ method effectively traces commit histories in comparison to the line-level SZZ method, it offers potential advantages for certain projects that lack such scenarios. Therefore, it is crucial for us to investigate not only false positive commits but also tracking accuracy.

In this analysis, we consider the commits to be accurately tracked candidates if they contain the specific code that causes the bug and are modified by the corresponding bug-fixing commits. Based on previous studies [3], [16], [18], [20], [21], we only extract commits from the candidates that meet the following criteria as the correctly tracked commits.

- Import changes are not tracked [18]
- Comment changes are not tracked [3], [18]
- Commits were not related to a style change (e.g., variable name changes, indentation changes) [16], [18]
- Commits were not related to a refactoring [20], [21]

B. Result

Difference. Figure 5b shows that the line-level SZZ method induces 34 unique false-positive commits. We hypothesize that the token-level tracking can ignore cosmetic changes, such as source code formatting and whitespace modifications and reduce the number of such false-positive commits. Based on our manual checking, we found seven false-positive commits induced by source code formatting and two induced by whitespace modifications in the 34 false-positive commits. In contrast, the false-positive commits induced by the token-level SZZ method do not include any caused by source code formatting or whitespace modifications. Therefore, the token-level SZZ method provides benefits in addressing a specific type of false-positive commits, even though it may induce more false-positive commits overall.

Tracking accuracy. Table V displays the proportion of false-positive bug-inducing commits that occur as a result of accurately tracking the change histories from bug-fixing commits. The proportions are 0.265 for line-level tracking, 0.396 for token-level tracking, and 0.688 for commits regardless of line-level and token-level tracking. This result suggests that tracking changes by the token-level may provide more accurate tracking compared to tracking by the line-level. In fact, we discovered commits that are only identified through token-level tracking.

Interestingly, more than half (68%) of the false positive commits identified by both line-level and token-level tracking were a result of accurately tracking the change histories. This suggests that the code lines identified by both line-level and token-level tracking are more likely to be the lines that were initially modified by bug-fixing commits. Therefore, if the SZZ method can disregard the cases where it theoretically fails, utilizing both tracking methods (i.e., a hybrid method) would lead to improved tracking accuracy.

> The token-level SZZ method is capable of avoiding false-positive bug-inducing commits that occur in the line-level SZZ method due to source code formatting and whitespace modifications. Furthermore, tracking at the token level and the hybrid approach of token-level and line-level tracking are more accurate than tracking at the line level. In the future, if we prevent the commits in which the SZZ method theoretically fails, the hybrid approach would improve the SZZ performance.

VII. Lessons Learned

Based on our empirical analysis, we have identified the potential benefits and pitfalls of using the token-level SZZ method. Table VI summarizes the potential benefits and pitfalls for both line-level and token-level SZZ methods. The checkmark indicates that the benefit/pitfall has not been reported by previous studies. The references indicate the previous studies that at least argue for that benefit/pitfall, even if they do not empirically validate them.

Our empirical study revealed one new benefit when using line-level tracking and three new pitfalls when using token-level tracking for the SZZ method (Please check ✓). Additionally, we validated the two benefits and two pitfalls that were reported in previous studies. Initially, we deduced that token-level tracking would be entirely beneficial for the SZZ method. However, identifying these benefits and pitfalls provides valuable lessons on how to avoid pitfalls and improve the performance of the SZZ method while maximizing its benefits.

Below, we present lessons learned from utilizing token-level tracking in SZZ, based on its potential benefits and pitfalls.

A. Tracking Accuracy

Based on the results from Table V, token-level tracking shows potential for accurately tracking change histories. Unlike line-level tracking, the token-level SZZ method can ignore cosmetic changes, which can improve tracking accuracy (TB_1 and LP_1). As a result, researchers and practitioners aiming to improve tracking accuracy can benefit from utilizing token-level tracking.

However, there is a pitfall in utilizing token-level tracking. Table II demonstrates that the token-level SZZ method performs worse in comparison to the line-level SZZ method. One reason for this is that when using token-level tracking, the blame operation in SZZ identifies a large number of bug-inducing commits, including numerous false positive bug-inducing commits, even if the tracking is accurate (TP_2). This is because token-level change histories consist of more targets (i.e., tokens) to track back the histories, as opposed to line-level change histories where the targets are lines.

Hence, it is important for researchers and practitioners not only to employ token-level tracking but also to suggest a filtering approach to mitigate the occurrence of false positives. For instance, our manual analysis revealed that tracking specific tokens such as parentheses (e.g., {}, ()) and semicolons often results in false-positive bug-inducing commits (TP_3). By implementing a filtering approach to exclude these tokens, the overall performance of the token-level SZZ method can be enhanced.

Lesson 1: The token-level SZZ method can improve tracking accuracy, but it does not enhance the overall performance. To improve the overall performance, it is important to propose a filtering approach that reduces the occurrence of false-positive bug-inducing commits.

B. Hybrid Approach

As explained in Section VII-A, utilizing token-level tracking with a filtering approach can help mitigate pitfalls. However, a combination of token-level and line-level tracking is also effective in addressing them. In fact, Table VI demonstrates that a pitfall encountered with token-level tracking (TP_1, MG2 at token-level [28]) can be resolved with line-level tracking (LB_1). Similarly, a pitfall encountered with line-level tracking (LP_1) can be addressed with token-level tracking (TB_1). For example, while token-level tracking may not capture commits that solely involve adding tokens (e.g., Listing 2), line-level tracking can track these commits.

A potential approach is for researchers and practitioners to transition from tracking at the token-level to tracking at the line-level when the target commits only involve adding tokens (LB_1 relieves TP_1). In contrast, they can transition from tracking at the line-level to tracking at the token-level when the target commits involve numerous cosmetic changes (TB_1 relieves LP_1). This approach allows them to effectively address the respective pitfalls.

It is worth noting that there is a pitfall that causes worse performance for both line-level and token-level tracking: when the target commits only involve added lines (LP_2), these commits can be categorized as *Mapping Ghost type 1 (MG1)* in the categorization defined by the previous study [28]. To address this challenge, researchers and practitioners need

TABLE VI: Potential benefits and pitfalls of using line-level and token-level tracking for the SZZ method (The checkmark (✓) indicates that this benefit/pitfall has not been reported by previous studies)

Type	Aspects	Ref. number	Description	Related work
Token-level	Benefits	TB_1	It is able to ignore cosmetic changes (e.g., formatting changes and whitespace changes)	[13]
		TB_2	It is able to track changes to a part of the line (e.g., changing a token)	[13]
	Pitfalls	TP_1	It is unable to track commits that only add tokens.	✓
		TP_2	It potentially induces a large number of false-positive bug-inducing commits.	✓
		TP_3	It frequently leads to false positives by tracking parentheses (e.g., {}, ()) and semicolons.	✓
Line-level	Benefits	LB_1	It is able to track changes that only add tokens.	✓
	Pitfalls	LP_1	It tracks cosmetic changes (e.g., formatting changes and empty line changes).	[16], [18]
		LP_2	It is unable to track commits that only add lines.	[23] [28]

to prepare not only a hybrid approach but also alternative solutions to improve the accuracy of the SZZ method.

Lesson 2: A hybrid approach that combines token-level and line-level tracking has the potential to perform well by addressing their respective pitfalls (TP_1 and LP_1). However, there is still a challenge that negatively affects the performance of the SZZ method (LP_2). Further studies are needed to propose alternative solutions for this challenge.

VIII. Threats to Validity

A. Construct Validity

The main threat is the quality of the ground truth data. In this paper, we utilized the developer-informed oracle dataset as our ground truth data. While this dataset contains high-quality bug-inducing commits, it also includes a significant number of false-negative commits. False-negative commits refer to commits that are actually bug-inducing, but were not identified as such. This is because the dataset relies on developer-provided information to prevent false-positive commits. Similarly, the dataset may also include false-positive commits. As a result, there may be a bias in the experimental results. However, we conducted thorough manual validation of the identified commits to ensure their accuracy. This process minimizes the bias as much as possible.

B. External Validity

In this study, we specifically focus on Java projects within the developer-informed oracle dataset. It is important to note that the findings may not be applicable to other datasets or programming languages. However, our dataset consists of 68 Java projects, covering a wide range of software systems, and Java is the most widely used programming language in similar studies [3], [16]–[18], [20], [21] [31]–[33]. For future research, we aim to include additional datasets and programming languages to improve the applicability of our findings.

C. Internal Validity

In this study, there is a potential bias of our manual analysis when trying to understand the difference between the line-based and token-based SZZ methods. To mitigate this bias, the first two authors conduct manual checks. The validity of our experimental scripts also exists. To ensure their validity, we make them available online[8].

[8]https://zenodo.org/doi/10.5281/zenodo.10044842

IX. Conclusion

In this paper, we conducted a study to compare the performance of the token-level SZZ method with the traditional line-level SZZ method. We aim to evaluate the impact of token-level tracking on the identification of bug-inducing commits. We assessed the accuracy of identifying these commits and analyzed their characteristics.

The results indicate that the token-level SZZ method identified two new bug-inducing commits. However, it had lower accuracy compared to the line-level SZZ method. Our in-depth analysis revealed that token-level tracking helps reduce false-positive bug-inducing commits caused by source code formatting and whitespace modifications. Additionally, we found that a hybrid approach combining token-level and line-level tracking has the potential to improve the accuracy of tracking commit histories. Based on these findings, we summarized the potential benefits and pitfalls of both token-level and line-level SZZ methods.

Our summary has the potential to serve as the foundation for future research aimed at enhancing the performance of the SZZ method. One suggestion we have is to explore a hybrid approach that combines token-level and line-level tracking to effectively address their respective pitfalls.

Acknowledgment

This research was partially supported by JSPS KAKENHI Japan (Grant Numbers: JP22K17874, 21H03416 and JP23K11046).

References

[1] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 26–36.

[2] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When Does a Refactoring Induce Bugs? An Empirical Study," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 104–113.

[3] D. 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, 2017.

[4] "Updated NIST Software Uses Combination Testing to Catch Bugs Fast and Easy." [Online]. Available: https://www.nist.gov/news-events/news/2010/11/updated-nist-software-uses-combination-testing-catch-bugs-fast-and-easy

[5] M. Zhivich and R. K. Cunningham, "The Real Cost of Software Errors," *IEEE Security and Privacy*, vol. 7, no. 2, pp. 87–90, 2009.

[6] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[7] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 3–13.

[8] M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 17:1–17:24, 2018.

[9] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *Journal of Systems and Software*, vol. 195, no. C, 2023.

[10] 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, 2013.

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[13] D. M. German, B. Adams, and K. Stewart, "Cregit: Token-level blame information in git version control repositories," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2725–2763, 2019.

[14] X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat, "Mining Software Repositories for Accurate Authorship," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, pp. 250–259.

[15] F. Servant and J. A. Jones, "Fuzzy Fine-Grained Code-History Analysis," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 746–757.

[16] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006, pp. 81–90.

[17] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, "Mining version archives for co-changed lines," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, pp. 72–75.

[18] C. Williams and J. Spacco, "SZZ revisited: Verifying when changes induce fixes," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, 2008, pp. 32–36.

[19] 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, no. 1, pp. 107–139, 2014.

[20] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the SZZ algorithm: An empirical study," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 380–390.

[21] E. C. Neto, D. A. D. Costa, and U. Kulesza, "Revisiting and Improving SZZ Implementations," in *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2019, pp. 1–12.

[22] P. Bludau and A. Pretschner, "PR-SZZ: How pull requests can support the tracing of defects in software repositories," in *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2022, pp. 1–12.

[23] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, "A comprehensive evaluation of SZZ Variants through a developer-informed oracle," *Journal of Systems and Software*, vol. 202, no. C, 2023.

[24] L. Tang, L. Bao, X. Xia, and Z. Huang, "Neural SZZ Algorithm," in *Proceedings of 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023, pp. 1024–1035.

[25] "jpace/diffj." [Online]. Available: https://github.com/jpace/diffj

[26] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, pp. 269–279.

[27] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 483–494.

[28] C. Rezk, Y. Kamei, and S. McIntosh, "The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3297–3309, 2022.

[29] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models," in *Proceedings of the 37th International Conference on Software Engineering*, vol. 1, 2015, pp. 812–823.

[30] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 392–401.

[31] V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri, "OpenSZZ: A Free, Open-Source, Web-Accessible Implementation of the SZZ Algorithm," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 446–450.

[32] 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*, 2019, pp. 7–12.

[33] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.

[34] M. Kondo, Y. Kashiwa, Y. Kamei, and O. Mizuno, "An empirical study of issue-link algorithms: Which issue-link algorithms should we use?" *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–50, 2022.

[35] S. Qiu, D. M. German, and K. Inoue, "An Exploratory Study of Copyright Inconsistency in the Linux Kernel," *IEICE Transactions on Information and Systems*, vol. E104.D, no. 2, pp. 254–263, 2021.

[36] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *Journal of Systems and Software*, vol. 158, no. C, 2019.

[37] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[38] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How Different Are Different diff Algorithms in Git?" *Empirical Software Engineering*, vol. 25, no. 1, pp. 790–823, 2020.