MV-SZZ: An Empirical Study of a Majority Voting-Based SZZ Method

Inase Kondo, Masanari Kondo, *Member, IEEE*, Daniel M. German, Yasutaka Kamei, *Senior Member, IEEE*, and Yoshiki Higo, *Member, IEEE*

***Abstract*—The SZZ method identifies defect-inducing commits by tracing lines modified in defect-fixing commits back to the commits that introduced them. While this method is widely used, it may fail to identify defect-inducing commits that are untraceable at the line-level. To address this limitation, a previous study proposed a Token-SZZ method that tracks changes at the token-level. This approach converts a line-level Git history into a token-level history by decomposing each line into individual tokens. While this method is able to identify defect-inducing commits that the previous SZZ method misses, it also incorrectly identifies many commits as defect-inducing (false positives), resulting in decreased performance. To mitigate this issue, we propose *Majority Voting SZZ (MV-SZZ)*, which consists of two key features: an N*-token representation* of the Git history and a *majority voting mechanism*. The N-token representation expands on the token-level concept (where $N=1$) by instead using N ($N>1$) consecutive tokens as a single line. This allows the method to capture more context for defect identification. The majority voting mechanism identifies the most frequent candidate commit associated with the changed tokens, and selects it as the defect-inducing commit. This approach effectively filters out unrelated tokens and reduces false positives. We compared MV-SZZ with six SZZ methods on both the Developer-IO and Defects4J datasets. MV-SZZ achieved the highest F1 and F0.5 scores on both datasets (F1: 0.587 and 0.580; F0.5: 0.591 and 0.578). Interestingly, we found that the issue of increased false positives due to token-level tracking is not unique to token-level methods; rather, it is a general problem that arises in SZZ methods when more accurate tracking is applied. Moreover, we demonstrated that our majority voting mechanism serves as an effective filtering strategy for SZZ variants in general.**

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

Received 2 June 2025; revised 30 March 2026; accepted 23 April 2026. Date of publication 28 April 2026; date of current version 1 June 2026. This work was supported in part by the JSPS KAKENHI under Grant JP23K24823, Grant JP24H00692, Grant JP24K02921, and Grant JP25K03100; in part by Japan Science and Technology Agency (JST) as part of Adopting Sustainable Partnerships for Innovative Research Ecosystem (ASPIRE) under Grant JPMJAP2415; in part by Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship; and in part by the NSERC. Recommended for acceptance by T. Hall. *(Corresponding author: Inase Kondo.)*

Inase Kondo and Yoshiki Higo are with The University of Osaka, Osaka 565-0871, Japan (e-mail: i-kondou@ist.osaka-u.ac.jp; higo@ist.osaka-u.ac.jp).

Masanari Kondo and Yasutaka Kamei are with Kyushu University, Fukuoka 819-0395, Japan (e-mail: kondo@ait.kyushu-u.ac.jp; kamei@ait.kyushu-u.ac.jp).

Daniel M. German is with the University of Victoria, Victoria, BC V8P 5C2, Canada (e-mail: dmg@uvic.ca).

Digital Object Identifier 10.1109/TSE.2026.3688089

I. Introduction

DEFECTS in software are inevitable because humans develop software and they may make mistakes during the development process. To address this issue, developers engage in debugging, which is both costly and time-consuming [1], [2]. As a result, researchers have studied various approaches to reduce the cost of debugging, including defect analysis, defect prediction, and automated program repair [3], [4], [5], [6], [7], [8], [9], [10].

The SZZ method is the de facto standard for constructing defect datasets used in defect-related studies [6], [8], [11]. The SZZ method identifies defect-inducing commits that introduced defective code by analyzing the version control history (e.g., Git) and utilizing the `blame` function [12], [13]. The SZZ method first identifies the commits that fixed defects, referred to as defect-fixing commits. Once these defect-fixing commits are identified, the `blame` function is applied to the changed lines in the defect-fixing commits. The `blame` command displays the commit hash that last modified a given line of code. The SZZ method assumes that the commit responsible for introducing the changed line is the defect-inducing commit.

While the SZZ method is widely used, its accuracy in identifying defect-inducing commits remains limited, partly due to inherent limitations of the blame function [14], [15]. The `blame` function tracks changes only at the line level, not at a finer granularity within those lines. As a result, SZZ may incorrectly identify defect-inducing commits. For example, if a developer modifies only a comment in a defective line, this change does not affect the defect. However, the `blame` function incorrectly identifies the commit that changed the comment as a defect-inducing commit.

To address this limitation, Watanabe et al. [16] proposed Token-SZZ. This method converts the development history so that each token appears on a separate line, allowing changes to be tracked at the token level. This token-level development history allows the `blame` function to trace changes at a token-level granularity. They reported that Token-SZZ can identify defect-inducing commits overlooked by the line-level SZZ method. While this method has the potential to improve the accuracy of the SZZ method, it also increases the number of incorrectly identified defect-inducing commits by tracking tokens unrelated to defects, such as parentheses. Hence, it is necessary to further investigate its advantages and disadvantages, and to

propose a method that maintains its benefits while mitigating its drawbacks.

In this study, we address the limitations of Token-SZZ and propose a new approach. We first replicate the study of Token-SZZ proposed by Watanabe et al. [16] to understand its strengths and weaknesses. We then propose a *majority voting SZZ method (MV-SZZ)* to address the weaknesses of Token-SZZ.

The MV-SZZ can be briefly summarized as follows:

1) Each original source code line is converted into multiple lines, with each line corresponding to a sliding window of N consecutive tokens (the current token plus $N-1$ following tokens). We call this the N-token representation of the source code. When $N=1$ this process is equivalent to Token-SZZ [16].
2) For a given value of N, the original git repository is converted into its N-token representation git repository.
3) The basic SZZ method is then applied to the N-token representation git repository, and the defect-inducing commit is selected via a majority voting mechanism.
4) These three steps are repeated with increasing values of N until a stopping criterion is met, at which point the optimal value of N is determined. The output of the MV-SZZ method is then obtained using this optimal value.

We evaluated the effectiveness of the proposed method using the Developer-informed oracle and Defects4J datasets, which include 79 OSS projects in total. To evaluate how the fundamental components of the MV-SZZ method (specifically, its N-token representation and majority voting mechanism) work, and to compare the MV-SZZ with existing SZZ methods, we addressed the following research questions:

RQ1: **How does applying the SZZ method to an N-token representation repository compare with applying it to the original repository?**
Results: Compared to the basic SZZ method, the N-token representation at $N=1$ identified fewer defect-inducing and incorrect commits. In contrast, when $N \geq 2$, the N-token representation identified more defect-inducing commits (up to 22) but also led to more incorrectly identified defect-inducing commits (up to 130).

RQ2: **How does the selection of defect-inducing commits by majority voting affect the SZZ method results?**
Results: Majority voting significantly reduces the number of incorrectly identified defect-inducing commits, while maintaining the number of correctly identified defect-inducing commits. For example, in the Developer-informed oracle dataset, when $N=5$, majority voting eliminates 101 incorrect identifications, with only eight correct identifications being affected.

RQ3: **Does the MV-SZZ method outperform the existing SZZ methods?**
Results: The MV-SZZ achieves the best performance in terms of F1 and F0.5 scores compared to the existing SZZ methods.

RQ4: **Does majority voting act as an effective filter to improve SZZ performance?**
Results: The majority voting mechanism functions as an effective filter, improving Precision, F1-score, and F0.5-score for the existing SZZ methods by reducing FPs.

The contributions of this study are as follows:

- We conduct a comparative study involving six existing SZZ methods. We find that applying more accurate tracking of development history increases the number of false positives produced by SZZ methods.
- We propose a new SZZ method called Majority Voting SZZ (MV-SZZ), and demonstrate that it outperforms existing SZZ methods by combining the N-token representation and majority voting.
- We investigate the impact of different N values on the accuracy of the SZZ method.
- We demonstrate that the majority voting mechanism can be applied as an effective filter not only to the N-token representation but also to the existing SZZ methods to improve their accuracy.
- To encourage further research, we provide a replication package[1] for the MV-SZZ method, including the implementation and datasets.

II. Background

This section introduces the SZZ method, reviews related work, outlines its limitations, and defines key terms.

A. SZZ Method

Sliwerski et al. [17] proposed the SZZ method (*B-SZZ*), which automatically identifies defect-inducing commits in a Git history. B-SZZ comprises two steps: (1) identifying defect-fixing commits and (2) identifying defect-inducing commits.

In the first step, a defect is selected from an issue tracking system, and its corresponding defect ID (e.g., the issue report number in JIRA[2]) is recorded. In practice, this process is performed for every defect, but for illustrative purposes, we focus on a single defect. The next task is to scan commit messages for the defect ID. Commits whose messages contain the defect ID are then classified as defect-fixing commits.

In the second step, B-SZZ applies the `blame` command to every line deleted by the defect-fixing commit. This command returns the commit hash corresponding to the last modification of each deleted line. These commit hashes are regarded as defect-inducing commits. The rationale is that the commits responsible for the deleted lines in defect-fixing commits are likely to have introduced the defect, as those lines are assumed to be directly related to the defect.

B. Related Work

While B-SZZ is widely used, its accuracy in identifying defect-inducing commits is limited. To address this issue, previous studies proposed various SZZ method variants [15], [16],

[1] https://zenodo.org/records/17945172
[2] https://www.atlassian.com/software/jira

[18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. Below, we summarize several representative SZZ variants.

AG-SZZ [14]: Kim et al. [14] reported that some deleted lines in a defect-fixing commit may be unrelated to the defect. For example, code formatting changes do not affect program behavior. To address this challenge, they proposed AG-SZZ, which leverages an annotation graph [18] to filter out deleted lines that do not impact code behavior, such as those involving whitespace and comments, thereby improving the accuracy of identifying defect-inducing commits.

MA-SZZ [20]: Da Costa et al. [20] identified a limitation of AG-SZZ where it mistakenly identified commits with only "meta-changes" as defect-inducing commits. "Meta-changes" include modifications like branch changes, merge changes, and "property changes", which refer to non-functional modifications like comments or code formatting. MA-SZZ solves this by modifying the annotation graph to connect all meta-change nodes to their prior changes.

L-SZZ, R-SZZ [15]: Davies et al. [15] proposed two variants to refine the defect-inducing commits identified by AG-SZZ. Specifically, these variants aim to filter out false-positive commits, that is, commits incorrectly identified as defect-inducing even though they did not actually introduce the defect. L-SZZ designates the commit with the most code changes as the defect-inducing commit, whereas R-SZZ chooses the most recent commit. These heuristics help select a single candidate when multiple defect-inducing commits are identified for a single defect-fixing commit.

Other variants implemented in PySZZ [29]: Rosa et al. [29] conducted a comprehensive empirical comparison of SZZ variants and released the PySZZ tool for such evaluations. In addition to the representative variants discussed above, PySZZ also includes heuristics for handling added lines, such as A-SZZ [21] and DU-based variants. Although these heuristics can in principle be combined with other SZZ variants, their original evaluation suggested only limited improvements. Therefore, we do not include them in the scope of this study.

Neural-SZZ [26]: Tang et al. [26] addressed the low precision of existing SZZ methods by using deep learning to filter out lines unrelated to defects. They proposed Neural-SZZ, which employs a graph attention network and a learning-to-rank technique. Neural-SZZ ranks deleted lines based on their relevance to the defect and applies the `blame` command to only the top-ranked lines. This strategy effectively reduces noise from irrelevant lines, leading to higher precision and F1-scores compared to previous SZZ methods.

Token-SZZ [16]: Watanabe et al. [16] proposed Token-SZZ to address the limitation of line-level `blame`, which cannot capture fine-grained modifications within a line. Their method transforms a Git history into a token-level representation so that `blame` can trace individual token changes rather than entire lines. As a result, Token-SZZ can identify defect-inducing commits that may be overlooked by SZZ methods operating on the original line-level history.

For example, Fig. 1 illustrates an example where the B-SZZ method fails to track bug-inducing lines. Commit A is a bug-fixing commit that fixes a bug by replacing variable x with

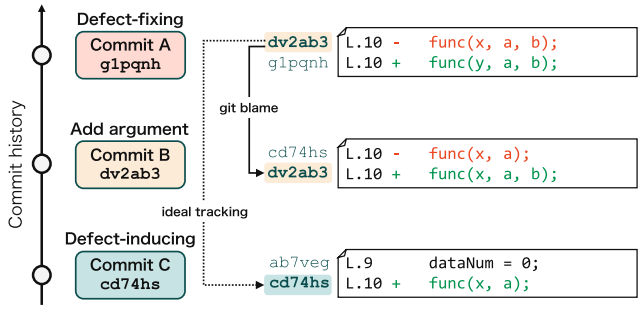

Fig. 1. Example of tracking an irrelevant replaced line in a defect-fixing commit.

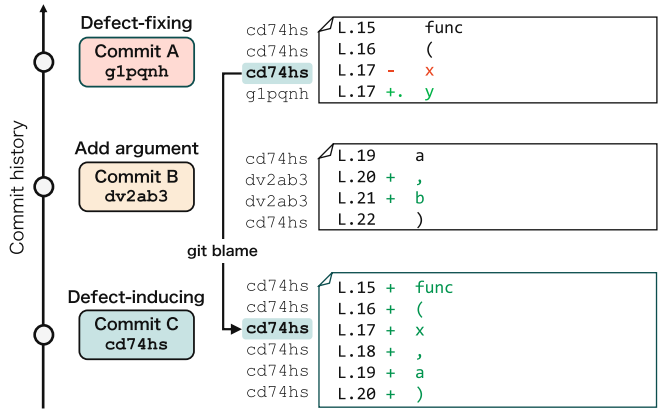

Fig. 2. Example of tracking a relevant replaced token in a defect-fixing commit using Token-SZZ.

y (i.e., the variable x was the cause of the bug). The B-SZZ method uses `blame` to track line 10 and identifies commit B as the defect-inducing commit. However, commit B only adds an argument to line 10. The actual bug-inducing commit is commit C, which introduces the argument x. Fig. 2 illustrates an example where Token-SZZ successfully identifies the bug-inducing commit. In this example, Token-SZZ tracks line 17, which contains the token x.

C. Limitation: Low Accuracy

While many variants of the SZZ method have been proposed to improve the identification of defect-inducing commits, their accuracy remains limited. For instance, although Token-SZZ can identify defect-inducing commits that the B-SZZ method overlooks, it also introduces false-positive defect-inducing commits, reducing the F1-score by 0.081 [16]. This indicates that even the most recent SZZ method variants still have limitations and that there is room for further improvement.

Hence, in this paper, we aim to improve the accuracy of the SZZ method. We build upon Token-SZZ, whose underlying concept is theoretically sound owing to its finer-grained representation of the Git history, and propose a new variant. Specifically, by adopting a finer-grained representation, we can track fine-grained changes within a line and identify defect-inducing commits that the B-SZZ method fails to detect. Therefore, we

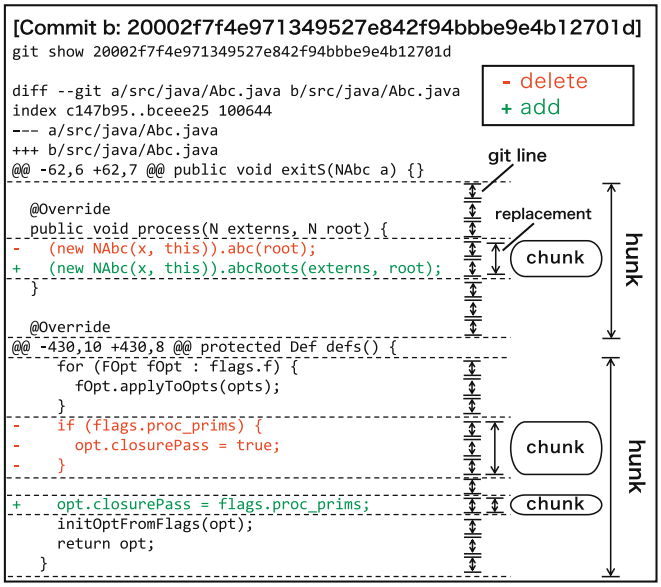

Fig. 3. Example of a git commit.

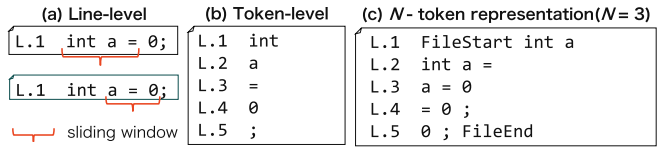

Fig. 4. Three types of git line representations.

TABLE I
PITFALLS OF TOKEN-SZZ REORGANIZED BASED ON THE PREVIOUS STUDY [16]

Ref. number	Description
PF1	It is unable to track commits when the changes in defect-fixing commits consist solely of adding tokens.
PF2	It has the potential to produce a large number of false-positive defect-inducing commits. For example, tracking very commonly used tokens, such as parentheses (e.g., {}, ()) and semicolons, can lead to such false positives.

believe that addressing these issues will lead to a significant improvement in accuracy.

D. Terminology

Git displays changes between commits as a sequence of *hunks*. Each hunk consists of *chunks*, which are sequences of lines. In a Git diff, these lines within a chunk are always marked as either added or deleted (which we refer to as *git lines*). Crucially, even a modification to an existing line is represented as a pair: the deletion of the original line and the addition of the new one. Fig. 3 illustrates an example commit containing two hunks and three chunks, where each chunk consists of consecutive git lines.

The SZZ method identifies defect-inducing commits by finding the original code segments that the defect-fixing commit modifies. This is achieved by examining the hunks of a defect-fixing commit to identify deleted lines, including those that were part of a modification. The underlying assumption is that deleted lines in a defect-fixing commit can be traced back to earlier changes that are candidates for the defect-inducing commit.

In this paper, we consider three git line representations: the existing line-level and token-level representations, and the proposed N-token representation. Fig. 4 illustrates these three representations. Fig. 4(a) depicts the line-level representation, where each git line corresponds to a source code line, and Fig. 4(b) illustrates the token-level representation, where each git line corresponds to a single token. Fig. 4(c) shows the N-token representation, where each git line consists of a sequence of N tokens. N specifies the number of tokens in each sequence. The N-token representation is implemented using a sliding window approach, where the source code is scanned and divided into consecutive sequences of N tokens. We refer to N as the *sliding window size*. In Fig. 4(c), the sliding window size is $N = 3$, and the window shifts by one token at a time, thereby converting the source code into the N-token representation.

Through this conversion, we reconstruct the repository such that each line corresponds to exactly one contiguous N-token sequence. By applying the standard `git blame` command to this reconstructed repository, we can trace the origin of these sequences. In other words, the traceability unit in our method is not an individual token, but the exact sequence of N tokens represented as a single git line.

In this study, we propose a new SZZ method based on the N-token representation. The following sections discuss the drawbacks of the token-level representation and explain how the N-token representation addresses these limitations.

III. PRELIMINARY STUDY

This preliminary study aims to replicate the recent variant of the SZZ method proposed by Watanabe et al. [16] and clarify its drawbacks. Given these drawbacks, we are motivated to refine Token-SZZ to improve its accuracy.

A. Motivation

Token-SZZ [16] intends to address the cases in which the defect-inducing line has been modified prior to the defect-fixing commit in areas not related to the defect. While this method improves the tracking of defect-inducing lines (benefits), it also introduces new challenges (pitfalls). Watanabe et al. [16] summarized such benefits and pitfalls by analyzing defect-inducing commits identified by both Token-SZZ and the traditional SZZ method. Table I reorganizes the reported pitfalls for the purpose of our analysis.

In this section, we replicate the study by Watanabe et al. [16] and investigate the pitfalls of Token-SZZ. Since the previous study was limited in terms of the studied SZZ methods and datasets, we extend their analysis to a large-scale empirical investigation. Specifically, we apply the same analysis to other

SZZ variants (AG-SZZ, MA-SZZ, L-SZZ, and R-SZZ) to examine whether the identified pitfalls are unique to Token-SZZ or common across different variants. Furthermore, because the original study evaluated only a single dataset, we include an additional dataset to enhance the external validity of the findings. Our goal is to confirm whether their findings are replicable and generalizable.

B. Approach

To replicate the study by Watanabe et al. [16], we use the same experimental settings to evaluate both Token-SZZ and the B-SZZ method. In addition to these two methods, we also evaluate other SZZ variants to investigate whether the pitfalls of Token-SZZ are specific to this method or apply generally to other SZZ variants. Specifically, we evaluate the accuracy of Token-SZZ and manually inspect the defect-fixing commits. Since Token-SZZ uses the information from the defect-fixing commits to identify the defect-inducing commits, we focus our manual inspection on the defect-fixing commits to clarify the causes of the pitfalls.

We compare Token-SZZ with the B-SZZ method and the other selected SZZ variants by examining the number of true positives (TP) and false positives (FP). Specifically, we analyze the TP and FP counts with respect to the number of deleted lines in the original commits.

The rationale behind this analysis is that Token-SZZ aims to improve the accuracy of tracking deleted lines using the `blame` command. We hypothesize that the pitfalls may be due to issues with tracking the deleted lines. To further investigate this, we categorize defect-fixing commits based on the number of deleted lines in the original commits and examine how this categorization affects the performance of Token-SZZ, the B-SZZ method, and the other selected SZZ variants. Based on this analysis, we manually investigate the defect-fixing commits to clarify the causes of the pitfalls.

We use the Developer-informed oracle dataset [29] and the Defects4J dataset [30]. In total, we use 79 OSS projects and 206 pairs of defect-fixing and defect-inducing commits.

C. Results

For commits with a small number of deleted lines, Token-SZZ consistently identifies fewer TP defect-inducing commits compared to the other SZZ methods. As shown in Fig. 5, which shows a bar plot with the vertical axis representing the number of TP defect-inducing commits, Token-SZZ detects fewer TPs than the other SZZ methods when one line is deleted. As shown in Table II, for the Developer-informed oracle dataset, Token-SZZ identifies five fewer TPs than the other SZZ methods. For the Defects4J dataset, it identifies at least two fewer TPs than the other SZZ methods. In contrast, for commits that delete three or more lines, all the compared methods yield a similar number of TPs.

To understand these differences, we manually examined defect-fixing commits, focusing on those that deleted only one line and where Token-SZZ failed to detect the corresponding defect-inducing commit that the other methods captured. In 14 out of 15 such cases, the defect was fixed solely by adding tokens; in other words, the commit fixed the defect without deleting any tokens. This situation corresponds to the first pitfall of Token-SZZ (PF1).

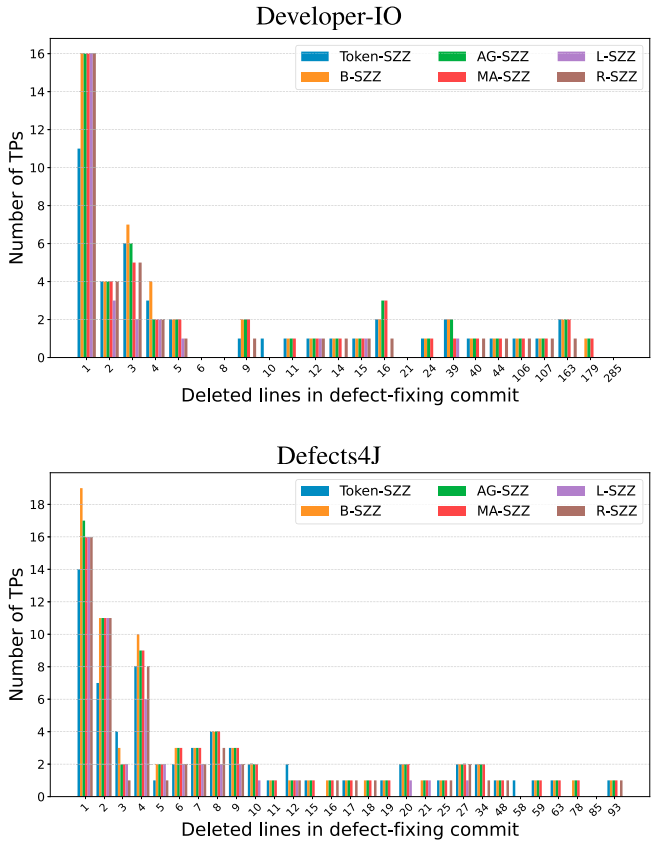

Fig. 5. Number of TPs for each deleted line in defect-fixing commits.

TABLE II
TP DIFFERENCE (TOKEN-SZZ – VARIANT) FOR DELETED-LINE = 1

Dataset	B-SZZ	AG-SZZ	MA-SZZ	L-SZZ	R-SZZ
Developer-IO	−5	−5	−5	−5	−5
Defects4J	−5	−3	−2	−2	−2

For example, Fig. 6 compares a line-level commit and a token-level commit. The line-level commit shows a deleted line, whereas the token-level commit does not. Consequently, Token-SZZ fails to identify the defect-inducing commit.

For commits with a small number of deleted lines in the Defects4J dataset, Token-SZZ identifies fewer FP defect-inducing commits compared to the other methods. As shown in Fig. 7, which shows a bar plot with the vertical axis representing the number of FP defect-inducing commits, Token-SZZ detects 28 fewer FPs than B-SZZ when between one and four lines are deleted in the Defects4J dataset.

For commits with a large number of deleted lines, not only Token-SZZ but also other SZZ variants tend to identify more FP defect-inducing commits compared to the B-SZZ method, except for L-SZZ and R-SZZ. Table III

```
22 -  if(lockFoodLevel){ +
22 if(lockFoodLevel && arena.inArena(p)){

78     if
79     (
80     lockFoodLevel
81 +   && +   arena +   . +   inArena +   ( +   p
       +   )
82     )
```

Fig. 6. Example commit that both adds and deletes a line in a line-level history, yet only adds seven tokens in a token-level history.

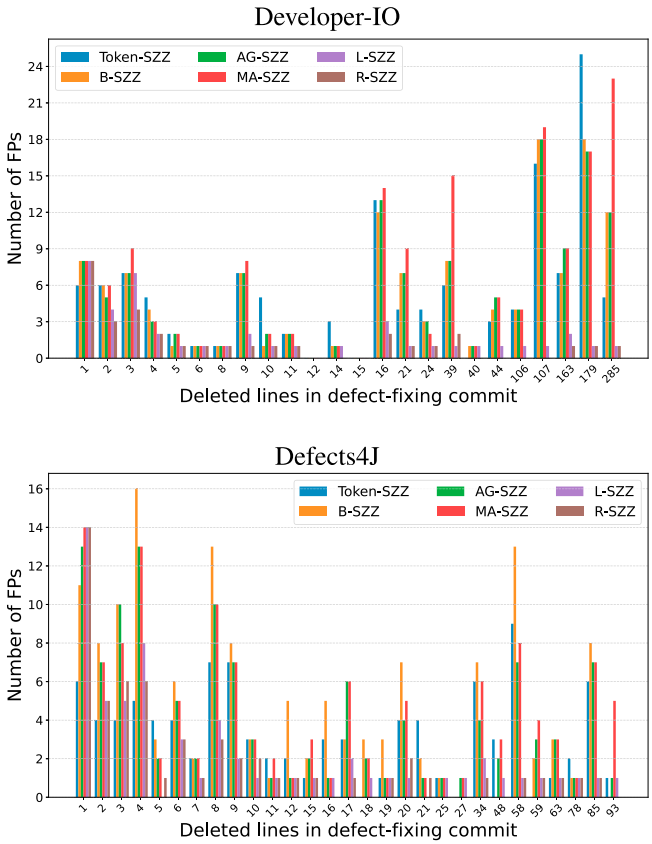

Fig. 7. Number of FPs for each deleted line in defect-fixing commits.

shows the FP difference compared to the B-SZZ method and highlights the cases where the number of FPs increases, particularly when a large number of lines are deleted. In the Developer-informed oracle dataset, Token-SZZ identifies 7 more FPs than B-SZZ for the defect-fixing commit with 179 deleted lines, and MA-SZZ identifies 11 more FPs for the defect-fixing commit with 285 deleted lines. In the Defects4J dataset, Token-SZZ, AG-SZZ, and MA-SZZ identify 3, 2, and 3 more FPs than B-SZZ, respectively, for the defect-fixing commit with 48 deleted lines. Hence, Token-SZZ, AG-SZZ, and MA-SZZ methods may occasionally increase the number of FPs when the number of deleted lines is large. In contrast, in such cases, L-SZZ and R-SZZ can substantially reduce the number of FPs.

TABLE III
FP DIFFERENCE (VARIANT – B-SZZ)

Dataset (deleted lines)	Token-SZZ	AG-SZZ	MA-SZZ	L-SZZ	R-SZZ
Developer-IO (179)	+7	−1	−1	−17	−17
Developer-IO (285)	−7	0	+11	−11	−11
Defects4J (48)	+3	+2	+3	+1	0

To understand the reasons for the increase in FPs, we manually examined defect-fixing commits. We observed that these commits removed entire lines of code, which often contain the cause of the defect but also include many other tokens that are unrelated to the defect. For example, tokens such as parentheses and semicolons are less likely to be the cause of defects and are less likely to be modified compared to other tokens. Unless the cause of the defect was introduced when the line was added, tracking parentheses and semicolons may lead to FPs. Consequently, Token-SZZ tracks these unrelated tokens, which can lead to FPs. This situation corresponds to the second pitfall of Token-SZZ (PF2).

PF2 is not unique to Token-SZZ but is a general pitfall applicable to SZZ variants that aim to improve tracking accuracy. Interestingly, we found that the second pitfall (PF2) can be applied to AG-SZZ and MA-SZZ since they also lead to an increase in FPs when a large number of lines are deleted. AG-SZZ and MA-SZZ are designed for more accurate tracking. We realized that this accurate tracking can increase FPs when they correctly track lines that are unrelated to the defect. For example, consider a defect-fixing commit that deletes two lines from a function unrelated to the bug, and this function had been moved from another file in a previous commit. B-SZZ might trace both lines back to the single file-move commit (resulting in one FP). In contrast, MA-SZZ correctly bypasses the file-move commit and traces the two lines back to the two different commits (resulting in two FPs).

This indicates that PF2 is not an exclusive drawback of the Token-SZZ but also applies to AG-SZZ and MA-SZZ. These three methods (Token-SZZ, AG-SZZ, and MA-SZZ) are all designed to improve tracking accuracy. Hence, PF2 is not unique to Token-SZZ but is a general pitfall applicable to SZZ variants that aim to improve tracking accuracy.

D. Discussion

Our preliminary study replicates the previous study [16] and reveals the same drawbacks. Given these drawbacks and our manual analysis, we propose two new solutions to improve Token-SZZ. First, we use an N-*token representation* per line, whereas Watanabe et al. [16] used a *single-token representation*. The single-token representation leads to missing cases in which a defect is fixed only by adding tokens (PF1). In contrast, the N-token representation can introduce deleted lines into the diff even in such cases. This is because adding one token changes the surrounding N-token sequences. This allows `blame` to trace the origin of those deleted lines. For example, Fig. 8 shows the 3-token representation of Fig. 6. While the single-token representation does not reveal any deleted tokens,

```
78      if  (  lockFoodLevel
79  -   (  lockFoodLevel  ) +  (
        lockFoodLevel  && +  lockFoodLevel &&
80  arena +  &&  arena  . +  arena  .
        inArena +  .  inArena  ( +
81  inArena  (  p +  (  p  ) +  p  )  {
82      )  {  return
```

Fig. 8. Commit diff in a 3-token representation format.

the 3-token representation shows deleted tokens at line 79. This representation enables Token-SZZ to identify the defect-inducing commit in cases where the single-token representation only shows added tokens.

Although the first solution may address PF1, it may also increase the number of deleted lines and lead to more FPs. Furthermore, our preliminary analysis shows that the original Token-SZZ also leads to more FPs (PF2). To address this issue, we use a *majority voting mechanism* as the second solution. This mechanism, similar to L-SZZ and R-SZZ, aims to reduce FPs by selecting the most suspicious defect-inducing commit from a set of candidate defect-inducing commits. Once all candidates for defect-inducing commits are identified for a defect-fixing commit, majority voting selects the candidate that is most frequently identified as the defect-inducing commit. The selected commit is then considered the defect-inducing commit corresponding to the defect-fixing commit. This method is expected to reduce FPs, which are often caused by tracking specific tokens (e.g., parentheses and semicolons) that are unlikely to be the majority among tokens in defect-fixing commits.

IV. Majority Voting SZZ: MV-SZZ

To address the drawbacks of Token-SZZ, we propose a new approach called *Majority Voting SZZ (MV-SZZ)*. MV-SZZ aims to prevent an increase in FPs while maintaining the number of TPs. This approach employs the N-token representation and the majority voting mechanism.

Fig. 9 shows an overview of MV-SZZ. The approach has four steps. First, MV-SZZ converts a Git history into an N-token representation. Second, the SZZ method is applied to the converted history to identify candidate defect-inducing commits. Third, MV-SZZ selects a defect-inducing commit by applying majority voting to the identified candidate defect-inducing commits. Fourth, N is incrementally increased until the stopping condition is met to determine the optimal value.

A. Create N-Token Representation History

MV-SZZ first converts the Git history of a target repository. Specifically, MV-SZZ applies `cregit` [31] to a Git repository, as in the previous study [16]. `cregit` is a tool that converts a Git history into a history containing a single-token representation per line. We modified `cregit` to convert a Git history into an N-token representation. The modified `cregit` has been included in our replication package. `Cregit` induces additional space and time costs during the conversion process.

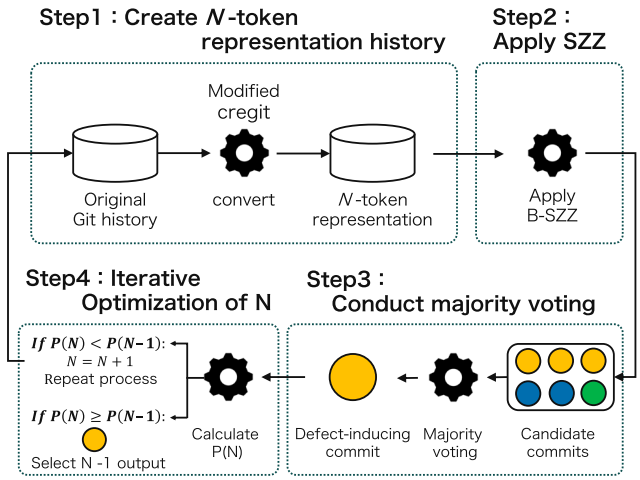

Fig. 9. Overview of the majority voting SZZ method.

In practice, these costs are manageable for typical repositories. The details of the costs are discussed in Sec. X-C.

B. Apply SZZ

MV-SZZ applies the B-SZZ method to the converted Git history. Although the B-SZZ method considers all candidates as defect-inducing commits, MV-SZZ makes no decision at this point; instead, it records all candidate defect-inducing commits. Note that since MV-SZZ uses the B-SZZ method, it is essential to have the complete N-token Git history to apply the `blame` command, the core component of the B-SZZ method, at the token level.

C. Conduct Majority Voting

Given the candidate defect-inducing commits, MV-SZZ selects the defect-inducing commit via majority voting. MV-SZZ counts the number of occurrences for each candidate within a defect-fixing commit and selects the candidate that appears most frequently. Fig. 10 shows our majority voting process. For a defect-fixing commit `wqg4bh` with 8 deleted lines, the B-SZZ method identifies three distinct defect-inducing commits. Among these, candidate `cbayg7` is identified for four deleted lines. As this candidate is the most frequently identified defect-inducing commit, MV-SZZ selects `cbayg7`. In the case of a tie, MV-SZZ selects all candidates that are most frequently identified.

D. Iterative Optimization of N

MV-SZZ includes a parameter N. The choice of N impacts the performance of MV-SZZ, but the optimal value of N may vary across different repositories. Hence, it is crucial to determine the optimal value of N for each target repository.

To optimize N, we designed an optimization process based on an assumption regarding the ideal behavior of SZZ methods. Specifically, we assume that an ideal SZZ method links each defect-fixing commit to exactly one defect-inducing commit.

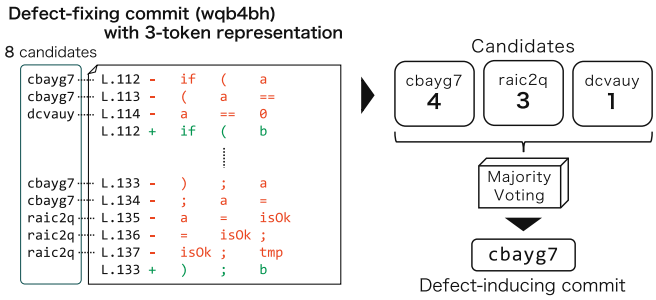

Fig. 10. Example of the majority voting process.

In other words, an ideal SZZ method would avoid the cases where a defect-fixing commit is linked to zero defect-inducing commits (*missed links*) or to multiple defect-inducing commits (*ambiguous links*). The rationale behind this assumption is that each defect-fixing commit typically addresses a specific defect, which is often associated with a single defect-inducing commit. Prior evidence supports this tendency: Lyu et al. [27] reported that 96.1% (73,109 out of 76,046) of Linux bug-fixing commits are linked to a single bug-inducing commit, and our inspection of the replication package of the developer-informed oracle dataset [29] showed a similar trend, with 99.75% (2,358 out of 2,364) of instances linked to a single bug-inducing commit. Therefore, the goal of the optimization process is to find the value of N that minimizes the number of missed and ambiguous links. Note that this assumption may not hold in all scenarios, such as when a defect-fixing commit addresses a defect introduced by multiple defect-inducing commits. Taken together, these observations suggest that the assumption is a practically reasonable basis for optimization in the majority of cases.

To evaluate an SZZ method based on this assumption, we quantify this objective by defining a *penalty score* $P(N)$ for each token granularity N. This score represents the total deviation from the ideal one-to-one mapping. The penalty score $P(N)$ is defined as follows.

$$P(N) = x_0 + \sum_{i=2}^{\infty} (i-1)x_i$$

x_i denotes the number of defect-fixing commits linked to i defect-inducing commits when using token granularity N. The ideal one-to-one links correspond to x_1 and incur no penalty, while other cases incur penalties. A heavier penalty is imposed when a defect-fixing commit is linked to multiple defect-inducing commits by multiplying x_i by $(i-1)$; x_0 (missed links) incurs a penalty of 1. A larger penalty score indicates a greater deviation from the ideal one-to-one mapping.

We iteratively evaluate MV-SZZ with increasing values of N from $N = 1$ and compute the corresponding penalty scores $P(N)$. We stop the iteration when we meet the *stopping condition* as follows.

$$P(N) \geq P(N-1)$$

This condition indicates that the penalty has either increased or remained the same when moving from $N-1$ to N. When this condition is met, it suggests that the current token granularity N is not providing any additional benefit over $N-1$, and we consider $N-1$ to be the optimal value.

V. Experimental Setup

This section outlines the experimental design for evaluating the effectiveness of the proposed MV-SZZ method. First, we introduce the research questions (RQs) and their motivation. Next, we provide an overview of the evaluation process. Finally, we describe each evaluation step in detail.

A. Research Questions

We define the following four RQs to evaluate the effectiveness of MV-SZZ:

RQ1: How does applying the SZZ method to an *N*-token representation repository compare with applying it to the original repository? We first investigate how the sliding window size N, a parameter of the MV-SZZ method, affects the performance of the SZZ method by comparing results across different values of N and against the B-SZZ method. In this RQ, we aim to clarify the characteristics of defect-inducing commits identified with different values of N and to evaluate how varying N impacts the accuracy of the SZZ method.

RQ2: How does the selection of defect-inducing commits by majority voting affect the SZZ method results? To reduce the number of FPs, we employ the majority voting mechanism for selecting defect-inducing commits. In this RQ, we investigate whether using majority voting affects the accuracy of MV-SZZ by comparing it with the B-SZZ method, both with and without the majority voting mechanism.

RQ3: Does the MV-SZZ method outperform the existing SZZ methods? We evaluate the performance of the MV-SZZ method in comparison with existing SZZ methods. This RQ aims to clarify both the effectiveness of the N-token representation and the majority voting mechanism for identifying defect-inducing commits.

RQ4: Does majority voting act as an effective filter to improve SZZ performance? We evaluate the impact of the majority voting mechanism on existing SZZ methods. This RQ aims to clarify the generalizability of the majority voting mechanism for improving accuracy across SZZ methods.

B. Overview

Fig. 11 presents an overview of the evaluation process. The evaluation follows these steps:

[Step 1]: Prepare datasets. We use repositories from the Developer-informed oracle dataset [29] and Defects4J [30] for evaluation. These datasets provide ground-truth information on defect-fixing commits and defect-inducing commits.

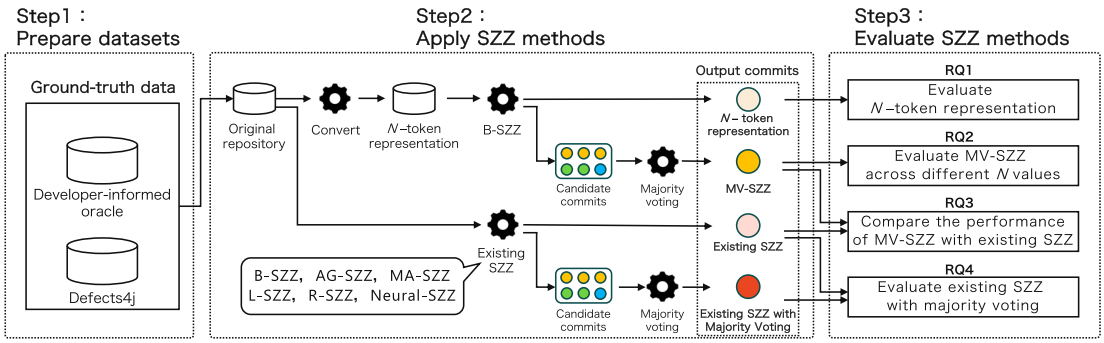

Fig. 11. Overview of the evaluation process.

TABLE IV
DATASET

Dataset	Project	Commit	Bugs
Developer-informed oracle dataset	68	997 287	76
Defects4J	11	26 239	130

[Step 2]: Apply SZZ methods. We apply our proposed method (MV-SZZ) and six existing SZZ methods (B-SZZ, AG-SZZ, MA-SZZ, L-SZZ, R-SZZ, and Neural-SZZ [14], [15], [17], [20], [26]) to the target repositories.

[Step 3]: Evaluate SZZ methods. We evaluate the impact of the value of N (RQ1), the effectiveness of majority voting (RQ2), the performance of the MV-SZZ method (RQ3), and the generalizability of majority voting (RQ4).

C. Datasets

To evaluate the effectiveness of SZZ methods, we use datasets that provide ground-truth information on defect-fixing and defect-inducing commits. In this study, we use the Developer-informed oracle dataset [29] and the Defects4J dataset [30] as our data sources (Table IV).

Developer-informed oracle dataset [29]. Rosa et al. [29] defined heuristics based on manual inspection to identify defect-inducing commits, resulting in a high-quality dataset of defect-fixing and defect-inducing commit pairs. The dataset includes 1,854 repositories written in eight major programming languages: C, C++, C#, Java, JavaScript, Ruby, PHP, and Python. In this study, we focus on the same subset of 68 Java projects analyzed by Watanabe et al. [16]. Following their methodology, we first filtered the 1,854 repositories and retained 72 repositories mainly developed in Java, because manual inspection of commits is required to assess the correctness of token-level tracking. This restriction to Java is also common in prior SZZ studies [14], [18], [19]. We then excluded four projects for the same reasons reported by Watanabe et al. [16]: one repository no longer existed, one bug-fixing commit had been removed, and two projects could not be processed by `cregit`. We use this exact subset because manual verification is equally crucial in our study. As a result, our final dataset consists of 68 projects with 997,287 commits, including 76 pairs of defect-fixing and defect-inducing commits.

Defects4J [30]. To enhance the generalizability of our study, we also utilize the Defects4J dataset as an additional data source. Defects4J provides a reliable list of defects and their corresponding defect-fixing commits, as it collects real-world defects introduced by human developers. However, Defects4J does not provide defect-inducing commits. Therefore, we used the pairs of defect-fixing and defect-inducing commits identified by Gabin et al. [32], [33] based on the Defects4J dataset. Specifically, their dataset consists of two parts: 67 bugs taken from an existing dataset constructed by Wen et al. [34] based on failing test cases, and 63 bugs manually curated by Gabin et al. [32], [33]. For the manually curated part, two authors identified the corresponding defect-inducing commits by consulting bug reports, failure symptoms, and developer patches. As a result, the final dataset used in this study consists of 130 pairs of defect-fixing and defect-inducing commits across 11 projects. We refer to this dataset as Defects4J.

D. Apply SZZ Methods

To evaluate the effectiveness of the proposed MV-SZZ method, we compare it with six existing SZZ methods: B-SZZ, AG-SZZ, MA-SZZ, R-SZZ, L-SZZ, and Neural-SZZ. B-SZZ, AG-SZZ, MA-SZZ, R-SZZ, and L-SZZ have been widely adopted as baselines in previous studies [26], [27], [28] and are implemented in the PySZZ v2 tool [29]. We apply these methods to the target repositories. In addition, we include Neural-SZZ as a state-of-the-art SZZ method for comparison and follow the implementation and experimental settings of the original study [26].

Additionally, we apply the MV-SZZ method to the same repositories. To assess the impact of the N-token representation on SZZ performance, we convert the target repositories into N-token representations using our modified version of the `cregit` tool [31].

E. Evaluation Metrics

We evaluate the accuracy of identifying defect-inducing commits for each SZZ method. Accuracy is assessed using six

metrics: TP, FP, Precision, Recall, F1-score, and F0.5-score. Precision is the proportion of actual defect-inducing commits among those identified as defect-inducing. Recall is the proportion of defect-inducing commits correctly identified among all actual defect-inducing commits. F1-score is the harmonic mean of Precision and Recall, providing a balanced evaluation of both metrics. These metrics have been widely used in previous studies [16], [26], [27] to evaluate the performance of SZZ methods. We also include the F0.5-score as an evaluation metric to better assess prediction accuracy [35]. F0.5-score is a weighted harmonic mean that emphasizes Precision more than Recall, and is therefore appropriate in our setting, where FPs are more costly than FNs. In the context of SZZ, FPs cause greater practical harm because they require developers to spend unnecessary manual effort inspecting incorrectly identified commits.

F. Statistical Analysis

To incorporate a statistical perspective into our comparison, we perform the Wilcoxon signed-rank test [36] on the paired distributions of the accuracy values (e.g., Precision). When a statistically significant difference is found ($p < 0.05$), we also compute Cliff's delta [37] to evaluate the effect size. This statistical comparison procedure, combining the per-commit scores with the Wilcoxon test and Cliff's delta, follows the approach of a previous study [29].

VI. RQ1. How Does Applying the SZZ Method to an N-Token Representation Repository Compare With Applying It to the Original Repository?

A. Motivation and Approach

RQ1 investigates the impact of the N-token representation on the accuracy of the SZZ method. Since N is a tunable parameter, we vary its value from 1 to 10 to clarify its effect on the SZZ method. We perform both quantitative and qualitative analyses. For the quantitative analysis, we evaluate the accuracy of the N-token representation for different values of N and compare the N-token representation with the B-SZZ method. In the qualitative analysis, we manually inspect the identified defect-inducing commits and their corresponding defect-fixing commits.

B. Results

When $N \geq 2$, the N-token representation outperforms the B-SZZ method in terms of TP, whereas it also results in an increase in FP. Table V shows the quantitative analysis results. The numbers in parentheses indicate the difference between the N-token representation and the B-SZZ method. Red numbers represent cases where the N-token representation outperforms the B-SZZ method, while blue numbers indicate cases where the B-SZZ method outperforms the N-token representation. Bold numbers highlight the best results across all N values.

We observe that increasing N from 1 to 9 leads to an increase in TPs across the two datasets. In particular, when $N \geq 2$, TPs exceed those of the B-SZZ method. This finding indicates that the N-token representation is capable of capturing defect-inducing commits that the B-SZZ method overlooks. However, the increase in N also leads to a significant rise in FPs. For example, in the Defects4J dataset, when $N = 10$, the N-token representation identifies 22 more TPs than the B-SZZ method, but it also identifies 130 more FPs. Therefore, it is crucial to mitigate the increase in FPs while maintaining the improvement in TPs to enhance the overall accuracy of the SZZ method.

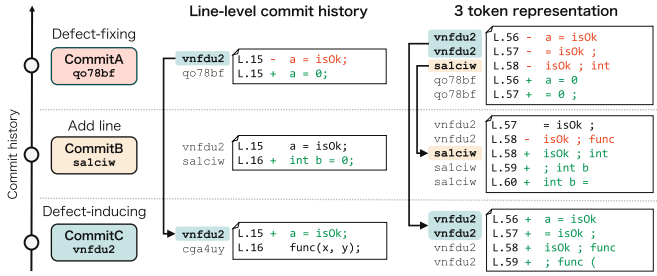

Fig. 12. 3-token representation increases the number of deleted-lines that are unrelated to the defect-inducing commit.

In general, the N-token representation does not outperform the B-SZZ method in terms of F1-score. Compared to the B-SZZ method, the N-token representation achieves a higher F1-score only when N is 1, 2, 3, and 4 in the Defects4J dataset. Specifically, the F1-score values for the N-token representation exceed those of the B-SZZ method by 0.005, 0.042, 0.040, and 0.028, respectively. These differences are not significant enough to conclude that the N-token representation outperforms the B-SZZ method in terms of accuracy.

The N-token representation increases the number of deleted lines that are unrelated to the defect-inducing commit, leading to an increase in FP. To understand the reason behind the increase in FPs, we manually analyzed the identified defect-inducing commits and their corresponding defect-fixing commits. We found that the inclusion of deleted lines not related to the defect-inducing commit is a significant factor contributing to the higher number of FPs.

For example, Fig. 12 shows a conceptual case, observed in our manual analysis, where FP increases. There are three commits: Commit A, Commit B, and Commit C. Commit A is a defect-fixing commit, and Commit C is a defect-inducing commit. Commit B is an unrelated commit. In the original Git history, Commit A contains one deleted line and one added line, and the deleted line correctly links to the defect-inducing commit (Commit C, `vnfdu2`).

However, in the 3-token representation, there are three deleted lines (lines 56–58). Two of these deleted lines (lines 56 and 57) correctly link to the defect-inducing commit (Commit C, `vnfdu2`), while one deleted line (line 58) links to an unrelated commit (Commit B, `salciw`). As a result, the SZZ method incorrectly identifies Commit B as a defect-inducing commit, leading to an increase in FP. Note that while `cregit` generates outputs in the form of `"token kinds | token"`, we illustrate only the tokens here to simplify the visualization.

TABLE V
COMPARISON OF THE PERFORMANCE OF THE B-SZZ METHOD AND THE N-TOKEN REPRESENTATION WITH $N = 1$ TO 10

	Developer-Informed-Oracle						Defects4J					
Method	TP	FP	Precision	Recall	F1 Score	F0.5 Score	TP	FP	Precision	Recall	F1 Score	F0.5 Score
B-SZZ	51	133	**0.277**	0.671	**0.392**	**0.314**	80	146	0.354	0.615	0.449	0.387
1-token	43 (−8)	**131** (−2)	0.247 (−0.030)	0.566 (−0.105)	0.344 (−0.048)	0.278 (−0.035)	66 (−14)	**95** (−51)	**0.410** (+0.056)	0.508 (−0.107)	0.454 (+0.005)	0.426 (+0.040)
2-token	52 (+1)	157 (+24)	0.249 (−0.028)	0.684 (+0.013)	0.362 (−0.030)	0.285 (−0.029)	83 (+3)	125 (−21)	0.399 (+0.045)	0.638 (+0.023)	**0.491** (+0.042)	**0.431** (+0.044)
3-token	53 (+2)	164 (+31)	0.244 (−0.033)	0.697 (+0.026)	0.365 (−0.027)	0.280 (−0.033)	91 (+11)	151 (+5)	0.376 (+0.022)	0.700 (+0.085)	0.489 (+0.040)	0.414 (+0.028)
4-token	55 (+4)	193 (+60)	0.222 (−0.055)	0.724 (+0.053)	0.340 (−0.052)	0.258 (−0.056)	95 (+15)	173 (+27)	0.354 (0.000)	0.731 (+0.116)	0.477 (+0.028)	0.395 (+0.008)
5-token	56 (+5)	197 (+64)	0.221 (−0.056)	0.737 (+0.066)	0.340 (−0.052)	0.257 (−0.057)	98 (+18)	211 (+65)	0.317 (−0.037)	0.754 (+0.139)	0.446 (−0.003)	0.359 (−0.028)
6-token	58 (+7)	205 (+72)	0.221 (−0.056)	0.763 (+0.092)	0.342 (−0.050)	0.257 (−0.057)	98 (+18)	233 (+87)	0.296 (−0.058)	0.754 (+0.139)	0.425 (−0.024)	0.337 (−0.050)
7-token	58 (+7)	216 (+83)	0.212 (−0.065)	0.763 (+0.092)	0.331 (−0.061)	0.247 (−0.067)	98 (+18)	251 (+105)	0.281 (−0.073)	0.754 (+0.139)	0.409 (−0.040)	0.321 (−0.066)
8-token	58 (+7)	224 (+91)	0.206 (−0.071)	0.763 (+0.092)	0.324 (−0.068)	0.241 (−0.073)	100 (+20)	261 (+115)	0.277 (−0.077)	0.769 (+0.154)	0.407 (−0.042)	0.318 (−0.069)
9-token	**59** (+8)	235 (+102)	0.201 (−0.076)	**0.776** (+0.105)	0.319 (−0.073)	0.236 (−0.078)	**102** (+22)	275 (+129)	0.271 (−0.083)	**0.785** (+0.170)	0.402 (−0.047)	0.311 (−0.076)
10-token	58 (+7)	240 (+107)	0.195 (−0.082)	0.763 (+0.092)	0.310 (−0.082)	0.229 (−0.085)	**102** (+22)	276 (+130)	0.270 (−0.084)	**0.785** (+0.170)	0.402 (−0.047)	0.311 (−0.076)

Answer to RQ1

When $N \geq 2$, the N-token representation improves TP (up to 22), but it also results in an increase in FP (up to 130). The N-token representation captures defect-inducing commits that the B-SZZ method overlooks, but it also includes deleted lines unrelated to the defect-inducing commit, leading to an increase in FP. To improve the accuracy of the SZZ method, it is essential to mitigate the increase in FP while preserving the improvement in TP.

VII. RQ2. How Does the Selection of Defect-Inducing Commits by Majority Voting Affect the SZZ Method Results?

A. Motivation and Approach

RQ2 investigates the impact of majority voting on the accuracy of the SZZ method. We conduct both quantitative and qualitative analyses. For the quantitative analysis, we first execute the B-SZZ method, as well as a version of B-SZZ with majority voting. The B-SZZ method with majority voting is a basic SZZ approach that uses majority voting to filter out incorrectly identified defect-inducing commits. This method demonstrates the impact of applying majority voting on the accuracy of the SZZ method. We consider these two methods as baselines. Additionally, we execute the MV-SZZ method, which uses the N-token representation and majority voting. We compare the performance of the MV-SZZ method with both the B-SZZ method and the B-SZZ method with majority voting across different values of N.

For the qualitative analysis, we manually inspect the commits that are either filtered out or retained by the majority voting process. We aim to understand the reasons behind the majority voting decisions.

B. Results

Using majority voting consistently improves FPs, Precision, F1-score, and F0.5-score compared to the B-SZZ method. Table VI shows the quantitative analysis results, displayed in the same format as in RQ1. The analysis indicates that applying majority voting significantly reduces FPs, whereas TPs remain relatively unchanged. For example, in the Developer-informed oracle dataset, when $N = 5$, the MV-SZZ method reduces FPs by 101, while TPs decrease by only eight, compared to the B-SZZ method.

In terms of Precision, F1-score and F0.5-score, majority voting leads to a notable improvement over the B-SZZ method. For example, the B-SZZ method with majority voting achieves F1-scores of 0.549 and 0.486, while the B-SZZ method achieves 0.392 and 0.449 in the Developer-informed oracle dataset and Defects4J, respectively. Hence, we conclude that using majority voting consistently results in better performance than the B-SZZ method in terms of FPs, Precision, F1-score, and F0.5-score.

When $N \geq 2$, the MV-SZZ method outperforms the B-SZZ method with majority voting. Compared to the B-SZZ method with majority voting, the MV-SZZ method with $N \geq 2$ achieves a higher F1-score, as shown in Table VI. For instance, in the Developer-informed oracle dataset, the MV-SZZ method with 4 tokens achieves the best F1-score of 0.587, while the B-SZZ method with majority voting achieves an F1-score of 0.549. In Defects4J, the MV-SZZ method with 10 tokens achieves the best F1-score of 0.591, compared to 0.486 for the B-SZZ method with majority voting. This implies that using the N-token representation with majority voting is more effective than using the line-level SZZ method with majority voting.

The MV-SZZ method outperforms the B-SZZ method by capturing deleted tokens that the B-SZZ method overlooks. To understand the reasons behind this improvement, we manually analyze the TPs detected by the MV-SZZ method but not by the B-SZZ method. Our analysis reveals that the N-token representation can capture changes in the sequence of adjacent lines of code. An N-token representation can merge tokens from different lines in the line-level history into a single line, representing that the lines containing those tokens are adjacent. Consequently, even when a commit only adds lines (without any deletions at the line level), the N-token representation can capture changes in the adjacent line sequence.

An example is shown in Fig. 13, where we display both a line-level commit and its 4-token representation. Developers inserted new lines (L23–L25) in this commit. Before this commit, lines 22, 26, and 27 appeared consecutively in the line-level history (i.e., an adjacent line sequence). The 4-token representation captures this relationship between lines 22 and 26. When new lines (L23–L25) were inserted, the adjacency between lines 22 and 26 disappeared. This change is reflected

TABLE VI
COMPARISON OF THE PERFORMANCE OF THE B-SZZ METHOD AND THE MV-SZZ METHOD WITH $N = 1$ TO 10

Method	Developer-IO						Defects4J					
	TP	FP	Precision	Recall	F1 Score	F0.5 Score	TP	FP	Precision	Recall	F1 Score	F0.5 Score
B-SZZ	**51**	133	0.277	**0.671**	0.392	0.314	**80**	146	0.354	**0.615**	0.449	0.387
B-SZZ with Majority Voting	42 (−9)	35 (−98)	0.545 (+0.268)	0.553 (−0.118)	0.549 (+0.157)	0.547 (+0.233)	62 (−18)	63 (−83)	0.496 (+0.142)	0.477 (−0.138)	0.486 (+0.037)	0.492 (+0.105)
MV-SZZ (1 token)	30 (−21)	**30** (−103)	0.500 (+0.223)	0.395 (−0.276)	0.441 (+0.049)	0.475 (+0.161)	54 (−26)	**37** (−109)	**0.593** (+0.239)	0.415 (−0.200)	0.489 (+0.040)	0.547 (+0.160)
MV-SZZ (2 tokens)	41 (−10)	31 (−102)	0.569 (+0.292)	0.539 (−0.132)	0.554 (+0.162)	0.563 (+0.249)	67 (−13)	51 (−95)	0.568 (+0.214)	0.515 (−0.100)	0.540 (+0.091)	0.556 (+0.169)
MV-SZZ (3 tokens)	43 (−8)	**30** (−103)	0.589 (+0.312)	0.566 (−0.105)	0.577 (+0.185)	0.584 (+0.270)	72 (−8)	58 (−88)	0.554 (+0.200)	0.554 (−0.061)	0.554 (+0.105)	0.554 (+0.167)
MV-SZZ (4 tokens)	44 (−7)	**30** (−103)	**0.595** (+0.318)	0.579 (−0.092)	**0.587** (+0.195)	**0.591** (+0.277)	74 (−6)	61 (−85)	0.548 (+0.194)	0.569 (−0.046)	0.558 (+0.109)	0.552 (+0.165)
MV-SZZ (5 tokens)	43 (−8)	32 (−101)	0.573 (+0.296)	0.566 (−0.105)	0.570 (+0.178)	0.572 (+0.258)	75 (−5)	60 (−86)	0.556 (+0.202)	0.577 (−0.038)	0.566 (+0.117)	0.560 (+0.173)
MV-SZZ (6 tokens)	44 (−7)	31 (−102)	0.587 (+0.310)	0.579 (−0.092)	0.583 (+0.191)	0.585 (+0.271)	76 (−4)	56 (−90)	0.576 (+0.222)	0.585 (−0.030)	0.580 (+0.131)	0.578 (+0.191)
MV-SZZ (7 tokens)	44 (−7)	33 (−100)	0.571 (+0.294)	0.579 (−0.092)	0.575 (+0.183)	0.573 (+0.259)	77 (−3)	60 (−86)	0.562 (+0.208)	0.592 (−0.023)	0.577 (+0.128)	0.568 (+0.181)
MV-SZZ (8 tokens)	43 (−8)	34 (−99)	0.558 (+0.281)	0.566 (−0.105)	0.562 (+0.170)	0.560 (+0.246)	76 (−4)	56 (−90)	0.576 (+0.222)	0.585 (−0.030)	0.580 (+0.131)	0.578 (+0.191)
MV-SZZ (9 tokens)	44 (−7)	32 (−101)	0.579 (+0.302)	0.579 (−0.092)	0.579 (+0.187)	0.579 (+0.265)	76 (−4)	56 (−90)	0.576 (+0.222)	0.585 (−0.030)	0.580 (+0.131)	0.578 (+0.191)
MV-SZZ (10 tokens)	44 (−7)	32 (−101)	0.579 (+0.302)	0.579 (−0.092)	0.579 (+0.187)	0.579 (+0.199)	78 (−2)	56 (−90)	0.582 (+0.228)	0.600 (−0.015)	**0.591** (+0.142)	**0.586** (+0.199)

Line-level commit history

```
L.21   if (!NodeUtil.isObjectLitKey(n, n.getParent())) {
L.22     ensureTyped(t, n, STRING_TYPE);
L.23 + } else {
L.24 +   // Object literal keys are not typeable
L.25 +   typeable = false;
L.26   }
L.27   break;
```

4-token representation

```
L.77   STRING_TYPE ) ; }
L.78 - ) ; } break
L.78 + ) ; } else
L.79 + ; } else {
L.80 + } else { comment
L.81 + else { comment typeable
L.82 + { comment typeable =
L.83 + comment typeable = false
L.84 + typeable = false ;
L.85 + = false ; }
L.86 + false ; } break
L.87   ; } break ;
```

Fig. 13. Defect-fixing commit `70f817` in Defects4J `google/closure-compiler`: the line-level diff shows only additions, while the 4-token representation indicates deletions.

as a deletion of the corresponding token line (red line 78) in the 4-token representation. This deleted token line allows the MV-SZZ method to successfully identify the defect-inducing commit, even though no lines were deleted at the line level.

Answer to RQ2

Majority voting consistently improves the SZZ method by reducing FPs and enhancing Precision, F1-score, and F0.5-score. When $N \geq 2$, the MV-SZZ method outperforms the B-SZZ method with majority voting, achieving higher F1-scores. The N-token representation enables the MV-SZZ method to capture deleted tokens that B-SZZ overlooks, thereby improving its accuracy.

VIII. RQ3. Does the MV-SZZ Method Outperform the Existing SZZ Methods?

A. Motivation and Approach

Previous studies have proposed various SZZ methods. In this RQ, we evaluate the performance of the MV-SZZ method against existing SZZ methods. We aim to determine whether the MV-SZZ method outperforms existing SZZ methods. B-SZZ, AG-SZZ, MA-SZZ, R-SZZ, L-SZZ, and Neural-SZZ are employed as existing SZZ methods for comparison. In addition to these methods, this comparison includes the B-SZZ method with majority voting analyzed in RQ2.

To conduct this comparison, we first determined the optimal value of N for MV-SZZ on each dataset. We applied the iterative optimization process. We calculated the penalty scores $P(N)$ for each value of N on both datasets. Based on this process, $N = 4$ was selected for the Developer-informed oracle dataset, and $N = 6$ was selected for the Defects4J dataset. Therefore, in this comparison, we use $N = 4$ and $N = 6$ for the Developer-informed oracle and Defects4J datasets, respectively.

We compare these methods using the common evaluation metrics described in Sec. V-E. In addition, RQ3 includes a statistical comparison to assess whether the performance differences between MV-SZZ and the other SZZ methods are significant, following the procedure described in Sec. V-F.

B. Results

The MV-SZZ method generally outperforms the other SZZ methods in terms of F1 and F0.5 scores. Table VII compares the accuracy of the MV-SZZ method with that of other SZZ methods. The numbers in parentheses represent the difference between the MV-SZZ method and the existing SZZ methods. Blue numbers indicate cases where the MV-SZZ method outperforms the existing methods, while red numbers indicate cases where the existing methods outperform the MV-SZZ method. Bold numbers highlight the best results across all methods.

For both datasets, the MV-SZZ method achieves the highest F1 and F0.5 scores among all SZZ methods. For example, in the Developer-informed oracle dataset, the MV-SZZ method outperforms the B-SZZ method with majority voting by 0.038 in F1-score and 0.044 in F0.5-score. In the Defects4J dataset, the MV-SZZ method improves F1-score by 0.020 and F0.5-score by 0.004 compared to the Neural-SZZ method. These improvements primarily result from the ability of the MV-SZZ method to significantly reduce FPs without substantially affecting the number of TPs.

MV-SZZ demonstrated a statistically significant superiority ($p < 0.05$) over B-SZZ, AG-SZZ, MA-SZZ, and L-SZZ. Table VII also provides the results of the Wilcoxon signed-rank test and Cliff's delta calculations between MV-SZZ

TABLE VII
COMPARISON OF THE PERFORMANCE OF THE MV-SZZ METHOD AND THE EXISTING SZZ METHOD

Method	Developer-IO ($N = 4$)						Defects4J ($N = 6$)					
	TP	FP	Precision	Recall	F1 Score	F0.5 Score	TP	FP	Precision	Recall	F1 Score	F0.5 Score
MV-SZZ	44	**30**	**0.595**	0.579	**0.587**	**0.591**	76	56	0.576	0.585	**0.580**	**0.578**
B-SZZ	**51** (+7)	133 (+103)	0.277* (−0.318)	**0.671** (+0.092)	0.392 (−0.195)	0.314* (−0.277)	**80** (+4)	146 (+90)	0.354** (−0.222)	**0.615** (+0.030)	0.449* (−0.131)	0.387** (−0.191)
AG-SZZ	49 (+5)	136 (+106)	0.265** (−0.330)	0.645 (+0.066)	0.375* (−0.212)	0.300** (−0.291)	76 (0)	121 (+65)	0.386** (−0.190)	0.585 (0.000)	0.465* (−0.115)	0.414** (−0.164)
MA-SZZ	47 (+3)	161 (+131)	0.226** (−0.369)	0.618 (+0.039)	0.331** (−0.256)	0.259** (−0.332)	75 (−1)	132 (+76)	0.362** (−0.214)	0.577 (−0.008)	0.445** (−0.135)	0.391** (−0.187)
L-SZZ	27 (−17)	43 (+13)	0.386** (−0.209)	0.355** (−0.224)	0.370** (−0.217)	0.379** (−0.212)	50 (−26)	63 (+7)	0.442** (−0.134)	0.385** (−0.200)	0.412** (−0.168)	0.430** (−0.148)
R-SZZ	38 (−6)	32 (+2)	0.543 (−0.052)	0.500 (−0.079)	0.521 (−0.066)	0.534 (−0.057)	56 (−20)	57 (+1)	0.496** (−0.080)	0.431** (−0.154)	0.461** (−0.119)	0.481** (−0.097)
Neural-SZZ	37 (−7)	35 (+5)	0.514* (−0.081)	0.487 (−0.092)	0.500* (−0.087)	0.508* (−0.083)	70 (−6)	**50** (−6)	**0.583** (+0.007)	0.538 (−0.047)	0.560 (−0.020)	0.574 (−0.004)
B-SZZ with Majority Voting	42 (−2)	35 (+5)	0.545 (−0.050)	0.553 (−0.026)	0.549 (−0.038)	0.547 (−0.044)	62 (−14)	63 (+7)	0.496** (−0.080)	0.477** (−0.108)	0.486** (−0.094)	0.492** (−0.086)

Note: * and ** indicate $p < 0.05$ and $p < 0.01$, respectively (Wilcoxon signed-rank test). Values with a single underline indicate negligible effect sizes, and values with a double underline indicate small effect sizes (Cliff's δ).

and the other SZZ methods. For R-SZZ and B-SZZ with majority voting, no statistically significant difference was found in the Developer-informed oracle dataset, but a significant difference ($p < 0.05$) was observed on the Defects4J dataset. Conversely, for Neural-SZZ, no statistically significant difference was observed on the Defects4J dataset, whereas a significant difference was found on the Developer-informed oracle dataset. In most cases where a significant difference was found, this significance was accompanied by a negligible effect size; however, the comparison against L-SZZ showed a small effect size.

Furthermore, the L-SZZ method and the R-SZZ method, which also employ a candidate commit selection step, show lower Precision, Recall, F1-score, and F0.5-score compared to the MV-SZZ method in both datasets. This suggests that, under the conditions of this study, the majority voting approach for selecting candidate commits used in the MV-SZZ method yields superior performance compared to the selection strategies in the L-SZZ method and the R-SZZ method.

Answer to RQ3

The MV-SZZ method achieved the highest F1 and F0.5 scores across both datasets. Furthermore, statistically significant differences were observed between MV-SZZ and all existing methods on at least one of the datasets.

IX. RQ4. DOES MAJORITY VOTING ACT AS AN EFFECTIVE FILTER TO IMPROVE SZZ PERFORMANCE?

A. Motivation and Approach

The Majority Voting (MV) mechanism improved the identification of defect-inducing commits in our earlier results. Because MV operates independently of how SZZ variants initially identify candidates, it can be easily incorporated into other existing approaches. Therefore, in RQ4, we investigate whether the benefits of MV generalize to other SZZ variants. If MV consistently improves performance across different methods, it can be recommended as a standard component for SZZ implementations.

We apply the MV mechanism to the candidate defect-inducing commits identified by the AG-SZZ and MA-SZZ methods and compare their performance with and without the MV mechanism. The evaluation follows the same metrics and statistical analysis procedures used in RQ3. Variants that already function primarily as filters, such as L-SZZ and R-SZZ, are excluded from this analysis because applying the MV mechanism to these methods would be redundant.

MV-SZZ consists of two components: the N-token representation and the MV mechanism. As with the MV mechanism, the N-token representation is also applicable to other SZZ variants. However, the primary focus of this study is to evaluate the effectiveness and generalizability of the MV mechanism itself. Therefore, combinations of the N-token representation with other SZZ variants are outside the scope of this paper and are left for future work.

B. Results

Applying the MV mechanism improves Precision, F1-score, and F0.5-score for both AG-SZZ and MA-SZZ. Table VIII presents the accuracy comparison between the original AG-SZZ and MA-SZZ methods and their counterparts with the MV mechanism applied. As shown by the Δ values in the table, applying MV substantially increases Precision (by +0.114 to +0.245) and the precision-focused F0.5-score (by +0.081 to +0.222). Although Recall decreases slightly, the F1-score also shows an overall improvement. This overall improvement stems primarily from the substantial reduction in FPs driven by the MV filter, which outweighs the minor loss in Recall.

The statistical analysis, detailed in Table VIII, provides further insight into the impact of the MV mechanism. For MA-SZZ, applying the MV filter resulted in a statistically significant improvement ($p < 0.05$) compared to the original method on both the Developer-informed oracle and Defects4J datasets. However, for AG-SZZ, the observed improvements did not reach statistical significance ($p \geq 0.05$) on either dataset. Regarding the magnitude of the impact, Cliff's delta calculations indicated that the effect sizes for the statistically significant improvements observed for MA-SZZ + MV were negligible on both datasets.

Answer to RQ4

The Majority Voting mechanism functions as a general filter improving Precision, F1-score, and F0.5-score for other SZZ methods (AG-SZZ, MA-SZZ) by reducing FPs. While these improvements reached statistical significance for MA-SZZ, the effect sizes were negligible. This suggests that the MV mechanism can function as a general filter, although its practical impact may vary across different SZZ methods.

TABLE VIII
IMPACT OF APPLYING MAJORITY VOTING (MV) COMPARED WITH ORIGINAL, AND STATISTICAL TEST RESULTS

Method	Developer-IO ($N = 4$)						Defects4J ($N = 6$)					
	TP	FP	Precision	Recall	F1 Score	F0.5 Score	TP	FP	Precision	Recall	F1 Score	F0.5 Score
AG−SZZ	40 (−9)	43 (−93)	0.482 (+**0.217**)	0.526 (−0.119)	0.503 (+**0.128**)	0.490 (+**0.190**)	62 (−14)	62 (−59)	0.500 (+**0.114**)	0.477 (−0.108)	0.488 (+**0.023**)	0.495 (+**0.081**)
MA−SZZ	40 (−7)	45 (−116)	0.471* (+**0.245**)	0.526 (−0.092)	0.497 (+**0.166**)	0.481* (+**0.222**)	61 (−14)	60 (−72)	0.504* (+**0.142**)	0.469 (−0.108)	0.486 (+**0.041**)	0.497 (+**0.105**)

Note: * and ** indicate $p < 0.05$ and $p < 0.01$, respectively (Wilcoxon signed-rank test). Values with a single underline indicate negligible effect sizes, and values with a double underline indicate small effect sizes (Cliff's δ).

TABLE IX
COMPARISON OF THE IMPACT OF TIE-HANDLING STRATEGIES (DEFECTS4J)

Strategy	TP	FP	Precision	Recall	F1	F0.5
excluding	73 (−3)	**51** (−5)	**0.589** (+0.013)	0.562 (−0.023)	0.575 (−0.005)	**0.583** (+0.005)
most-changed-lines	75 (−1)	**53** (−3)	**0.589** (+0.010)	0.577 (−0.008)	**0.581** (+0.001)	**0.584** (+0.006)
most-recent	73 (−3)	**55** (−1)	0.570 (−0.006)	0.562 (−0.023)	0.566 (−0.014)	0.569 (−0.009)

X. DISCUSSION

A. Handling Ties in Majority Voting

In RQ1 and RQ2, we examined how the sliding window size N and majority voting affect the accuracy of the SZZ method. We found that the optimal sliding window size, in combination with majority voting, improves the accuracy of the SZZ method. Based on these findings, we proposed the MV-SZZ method. While our MV-SZZ method outperforms existing SZZ methods (RQ3), there is still room for improvement. In this discussion, we take a closer look at the tie-handling strategy in the MV-SZZ method to investigate its impact on the accuracy of identifying defect-inducing commits. We aim to suggest a way to further improve the MV-SZZ method and encourage additional research in this area.

1) Approach: In the initial implementation of the MV-SZZ method, when a tie occurs, we select all tied commits. However, the handling of ties is a key design choice, and it is not self-evident which strategy yields the best accuracy. Therefore, in this discussion, we compare this selecting strategy with three alternative tie-handling strategies and assess their impact on accuracy. Specifically, we consider (1) *the excluding strategy*, which excludes all tied commits, (2) *the most-changed-lines strategy*, which selects the commit with the most changed lines among the tied commits, and (3) *the most-recent strategy*, which selects the most recent commit among the tied commits.

The parameter N was set to the same values as in RQ3. In this setting, we realized that ties only occurred in the Defects4J dataset; thus, we evaluated the accuracy of these strategies using only the Defects4J dataset.

The results are shown in Table IX. The left column displays the performance of the three alternative tie-handling strategies. The right column shows the performance difference between such alternative strategies and the selecting strategy. Bold values indicate cases where the alternative strategy outperforms the selecting strategy.

2) Results: **There is no single silver bullet among the tie-handling strategies.** Each strategy exhibits distinct strengths and weaknesses across evaluation metrics. The excluding strategy achieves the best Precision, improving it by 0.013 compared to the selecting strategy, indicating that discarding ambiguous candidates can effectively reduce FPs. The most-changed-lines strategy offers the highest F0.5-score improvement of 0.006 and a slight increase in F1-score of 0.001, suggesting that prioritizing commits with larger changes better balances precision-oriented performance. In contrast, the selecting strategy attains the highest Recall, reflecting its tendency to include more potential defect-inducing commits. **In summary, the optimal tie-commit strategy should be selected based on the metric prioritized for evaluation.**

B. MV-SZZ vs Neural-SZZ

In RQ3, we observed that MV-SZZ slightly outperformed Neural-SZZ in terms of F1 and F0.5 scores, while Neural-SZZ also demonstrated competitive performance. However, a key practical distinction lies in the way the two methods are configured. Neural-SZZ requires manually annotated training data and model training, whereas MV-SZZ does not rely on supervised training. This difference introduces two practical limitations for Neural-SZZ. First, preparing manually annotated training data requires substantial effort and time. Second, determining an appropriate training configuration, such as the number of training epochs, incurs additional cost because it directly affects model performance and generalization.

To further investigate the second limitation, we analyzed the training behavior of Neural-SZZ using the same manually annotated data as the original paper [26]. Fig. 14 illustrates the F1-score of Neural-SZZ at each epoch for both datasets. The results show that the optimal number of epochs differs across datasets: on Defects4J, performance peaked at epoch 6 and remained stable afterward, while on the Developer-informed oracle dataset, the best result was achieved at epoch 0, meaning that training actually degraded performance. These findings indicate that Neural-SZZ is prone to overfitting the training data and that identifying an appropriate training configuration requires non-trivial effort.

Overall, MV-SZZ provides practical advantages over Neural-SZZ. It avoids the need for manually annotated training data and model training, eliminates the cost of epoch tuning, and demonstrates slightly better performance.

C. Running Time and Space Costs

Running time and space costs are critical considerations when using SZZ methods in practice. Therefore, in this section, we discuss the running time and space costs associated with our MV-SZZ approach.

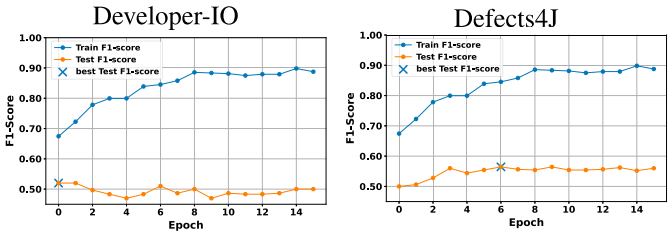

Fig. 14. F1-score of Neural-SZZ each epoch on the two datasets.

TABLE X
RUNNING TIME BY DATASET (MINS)

Dataset	MV-SZZ	B-SZZ	AG-SZZ	MA-SZZ	L-SZZ	R-SZZ	Neural-SZZ
Developer-IO	2348	4	13	16	19	15	17
Defects4J	155	1	4	6	7	6	21

The running time of MV-SZZ is considerably higher than the other studied SZZ variants. Table X shows the running time comparison among all studied SZZ methods. MV-SZZ takes approximately 123 to 587 times longer than the other SZZ variants on the Developer-informed oracle dataset, and 7.4 to 155 times longer on the Defects4J dataset. Interestingly, the total runtime for MV-SZZ on Defects4J is significantly shorter than on the Developer-informed oracle dataset.

The total running time of MV-SZZ is primarily dominated by two factors: (1) the execution cost of `cregit` itself, and (2) the repeated invocation of `cregit` during the iterative search for the optimal value of N. In our experiments, `cregit` accounts for 1,974 out of 2,348 minutes on the Developer-informed oracle dataset and 106 out of 155 minutes on the Defects4J dataset. Thus, improving the cregit-related part of the pipeline is essential for making MV-SZZ practical.

`Cregit` consists of two main steps: tokenization and writing the converted commit history. As noted in the original `cregit` paper [31], the tokenization step is the most time-consuming. However, `cregit` already performs tokenization using srcML [38], which provides an efficient source-code transformation infrastructure. Therefore, the more immediate opportunity for reducing the running time of MV-SZZ is not to redesign tokenization itself, but to reduce how often tokenization is invoked during MV-SZZ execution.

First, the execution scope of `cregit` can be reduced. Our current implementation processes the entire project history. However, MV-SZZ does not require all files in all commits. A practical optimization is to first identify the files changed by the defect-fixing commit, then trace only the history of those files (e.g., using file-restricted `git log` with `--follow` where applicable), and finally tokenize only the relevant files in the relevant commits. This optimization can substantially reduce both the number of processed commits and the number of tokenized files.

Second, the number of `cregit` executions can be reduced. In the current iterative process, `cregit` is executed independently for each value of N. However, once tokenization has been performed, the token stream can be reused to generate commit histories for multiple values of N. Since tokenization is the most expensive step, performing it only once and writing multiple N-token histories from the same tokenized output would substantially reduce the total running time.

These optimizations are promising engineering directions for improving the practical scalability of MV-SZZ, and we consider them important topics for future work. If implemented, they would directly address the main bottlenecks of the current implementation, especially the repeated invocation of `cregit` and its unnecessarily broad execution scope.

Regarding space costs, applying `cregit` generates a new N-token repository for each iteration. As these repositories contain significantly more lines than the original source code, they consume considerable storage space. For example, the original Defects4J dataset is 0.42 GB, but the generated N-token repositories grow to 1.71 GB during the process.

However, these space costs are temporary and manageable. The purpose of MV-SZZ is to identify defect-inducing commits. Once the SZZ analysis is performed for a specific N, the corresponding N-token repository is no longer needed. The repository can be deleted before proceeding to the next iteration, preventing a cumulative burden on storage.

XI. THREATS TO VALIDITY

Construct Validity: The validity of our study depends on the quality of the ground truth data. The Developer-informed oracle dataset [29] relies on information provided by developers, while the Defects4J dataset contains defect-inducing commits labeled by Gabin et al. [33]. The quality of these datasets may introduce bias into the experimental results. To address this, we incorporated manual verification in our experiments; however, manual verification does not cover the entire dataset. Future studies should conduct replication studies with other datasets to further validate our findings.

External Validity: Our experiments were conducted using Java projects from the Developer-informed oracle and the Defects4J datasets. The use of two datasets, each created through different methods, enhances the generalizability of our results. These datasets collectively include 79 distinct projects from various application domains and sizes, strengthening the generalizability of our findings within the Java ecosystem. However, as both datasets consist solely of Java projects, our findings may not directly apply to other programming languages. Future studies are encouraged to investigate the applicability of our findings to other programming languages.

Internal Validity: We implemented existing SZZ methods using the PySZZ library [29] and the Neural-SZZ replication package [26]. Consequently, the performance of these methods is dependent on these specific implementations.

Another potential threat is the exploratory manual analysis performed to identify the characteristics of defect-inducing commits identified by SZZ methods. This exploratory process may introduce subjectivity, as this process relies on the human judgment. To mitigate this threat, the analysis was conducted independently by two authors. For RQ1 and RQ2, the two

authors independently examined the target bug-fixing commits and discussed their qualitative characteristics. We report the agreement percentage as the proportion of bug-fixing commits for which the two authors initially reached the same interpretation before discussion. The resulting agreement percentage was 92/206 (44.7%) for RQ1 and 123/206 (59.7%) for RQ2. Because the analysis used open coding [39], the initial phase allowed flexibility in wording, granularity, and perspective, which likely contributed to the relatively low agreement percentages. After the independent analyses, we discussed the disagreements and reached a consensus on the final set of characteristics. In RQ1 and RQ2, we then reported the most comprehensive qualitative characteristics identified through these discussions.

XII. Conclusion

In this paper, we proposed and evaluated a novel SZZ method, called MV-SZZ. The MV-SZZ method consists of two main components: (1) the N-token representation, which converts each line-level git line into N consecutive tokens, and (2) a majority voting mechanism to select defect-inducing commits. We demonstrated that the N-token representation identifies more correct as well as more incorrect defect-inducing commits. We then showed that the majority voting mechanism significantly reduces the number of incorrect commits. We also found that the majority voting mechanism is effective not only for our N-token representation but also for existing SZZ methods. We evaluated MV-SZZ against six existing SZZ methods on the Developer-informed oracle and Defects4J datasets. The results showed that MV-SZZ achieved the best F1 and F0.5 scores among all methods.

The accuracy of MV-SZZ relies on the assumption that most lines deleted in defect-fixing commits are directly related to the defect injection. However, this assumption does not always hold true. For instance, a defect-fixing commit may include changes unrelated to the defect, such as modifications resulting from commit untangling [40]. These unrelated deleted lines can influence the vote, potentially causing a commit unrelated to the original defect to be mistakenly identified as the source. Therefore, the accuracy of MV-SZZ could be further improved by identifying and filtering out such unrelated lines before the majority voting process.

References

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Increasing software development productivity with reversible debugging." [Online]. Available: https://undo.io/media/uploads/files/Undo_Time-Travel-Debugging_UDBwhitepaper_20210111.pdf

[2] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Secur. Priv*, vol. 7, no. 2, pp. 87–90, Mar./Apr. 2009.

[3] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does a bug survive? An empirical study," in *Proc. 18th Work. Conf. Reverse Eng.*, 2011, pp. 191–200.

[4] C. Bird et al., "Fair and balanced? Bias in bug-fix datasets," in *Proc. 7th Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 121–130.

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

[6] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic program repair," *IEEE Softw.*, vol. 38, no. 4, pp. 22–27, Jul./Aug. 2021.

[7] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante, "Do developers introduce bugs when they do not communicate? The case of eclipse and Mozilla," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, 2012, pp. 139–148.

[8] Y. Kamei et al., "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[9] F. Servant and J. A. Jones, "Fuzzy fine-grained code-history analysis," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 746–757.

[10] C. L. 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 *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 3–13.

[11] R.-M. Karampatsis and C. Sutton, "ManySStuBs4J Dataset," Version 2.0, Zenodo, Feb. 2020, doi: 10.5281/zenodo.3653444.

[12] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar./Apr. 2008.

[13] S. Chacon and J. Long, "Git-git-blame documentation," 2024. Accessed: Apr. 28, 2025. [Online]. Available: https://git-scm.com/docs/git-blame

[14] E. J. Whitehead, Jr., "Automatic identification of bug-introducing changes," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 81–90.

[15] S. Davies, M. Roper, and M. Wood, "Comparing Text-Based and Dependence-Based Approaches for Determining the Origins of Bugs," *J. Softw.: Evol. Process*, vol. 26, no. 9, pp. 117–139, 2014.

[16] H. Watanabe, M. Kondo, E. Choi, and O. Mizuno, "Benefits and pitfalls of token-level SZZ: An empirical study on OSS projects," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, 2024, pp. 776–786.

[17] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[18] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, "Mining version archives for co-changed lines," in *Proc. Int. Workshop Mining Softw. Repositories*, 2006, pp. 72–75.

[19] C. Williams and J. Spacco, "SZZ revisited: Verifying when changes induce fixes," in *Proc. Workshop Defects Large Softw. Syst.*, 2008, pp. 32–36.

[20] 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 Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.

[21] E. Sahal and A. Tosun, "Identifying bug-inducing changes for code additions," in *Proc. 12th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2018, pp. 1–2.

[22] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the SZZ algorithm: An empirical study," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng.*, 2018, pp. 380–390.

[23] E. C. Neto, D. A. d Costa, and U. Kulesza, "Revisiting and improving SZZ implementations," in *Proc. ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2019, pp. 1–12.

[24] C. Rezk, Y. Kamei, and S. McIntosh, "The ghost commit problem when identifying fix-inducing changes: An empirical study of apache projects," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3297–3309, Sep. 2022.

[25] P. Bludau and A. Pretschner, "PR-SZZ: How pull requests can support the tracing of defects in software repositories," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, 2022, pp. 1–12.

[26] L. Tang, L. Bao, X. Xia, and Z. Huang, "Neural SZZ algorithm," in *Proc. 38th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2023, pp. 1024–1035.

[27] Y. Lyu, H. J. Kang, R. Widyasari, J. Lawall, and D. Lo, "Evaluating SZZ implementations: An empirical study on the Linux Kernel," *IEEE Trans. Softw. Eng.*, vol. 50, no. 9, pp. 2219–2239, Sep. 2024.

[28] S. Perez-Rosero, R. Dyer, S. W. Flint, S. McIntosh, and W. Srisa-An, "WIA-SZZ: Work item aware SZZ," 2024. [Online]. Available: https://arxiv.org/abs/2411.12740

[29] G. Rosa et al., "A comprehensive evaluation of SZZ variants through a developer-informed oracle," *J. Syst. Softw.*, vol. 202, Aug. 2023, Art. no. 111729.

[30] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal.*, 2014, pp. 437–440.

[31] D. M. German, B. Adams, and K. Stewart, "cregit: Token-level blame information in git version control repositories," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2725–2763, 2019.

[32] G. An and S. Yoo, "Reducing the search space of bug inducing commits using failure coverage," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2021, pp. 1459–1462.

[33] G. An, J. Hong, N. Kim, and S. Yoo, "Fonte: Finding bug inducing commits from failures," in *Proc. 45th Int. Conf. Softw. Eng.*, 2023, pp. 589–601.

[34] M. Wen et al., "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA: ACM, 2019, pp. 326–337.

[35] A. M. Ismail, S. H. A. Hamid, A. A. Sani, and N. N. M. Daud, "Toward reduction in false positives just-in-time software defect prediction using deep reinforcement learning," *IEEE Access*, vol. 12, pp. 47568–47580, 2024.

[36] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: http://www.jstor.org/stable/3001968

[37] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychol. Bull.*, vol. 114, no. 3, pp. 494–509, 1993. [Online]. Available: https://www.scopus.com/pages/publications/12044258480

[38] M. L. Collard, M. J. Decker, and J. I. Maletic, "SRCML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *Proc. IEEE Int. Conf. Softw. Maint.*, 2013, pp. 516–519.

[39] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociol.*, vol. 13, no. 1, pp. 3–21, 1990, doi: 10.1007/BF00988593.

[40] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Proc. 22nd Int. Conf. Softw. Anal., Evol. Reeng.*, 2015, pp. 341–350.