# Enhancing Bug-Inducing Commit Identification: A Fine-Grained Semantic Analysis Approach

Lingxiao Tang, Chao Ni, Qiao Huang, and Lingfeng Bao

***Abstract*—The SZZ algorithm and its variants have been extensively utilized for identifying bug-inducing commits based on bug-fixing commits. However, these algorithms face challenges when there are no deletion lines in the bug-fixing commit. Previous studies have attempted to address this issue by tracing back all lines in the block that encapsulates the added lines. However, this method is too coarse-grained and suffers from low precision. To address this issue, we propose a novel method in this paper called SEM-SZZ, which is based on fine-grained semantic analysis. Initially, we observe that a significant number of bug-inducing commits can be identified by tracing back the unmodified lines near added lines, resulting in improved precision and F1-score. Building on this observation, we conduct a more fine-grained semantic analysis. We begin by performing program slicing to extract the program part near the added lines. Subsequently, we compare the program's states between the previous version and the current version, focusing on data flow and control flow differences based on the extracted program part. Finally, we extract statements contributing to the bug based on these differences and utilize them to locate bug-inducing commits. We also extend our approach to fit the scenario where the bug-fixing commits contain deleted lines. Experimental results demonstrate that SEM-SZZ outperforms the state-of-the-art methods in identifying bug-inducing commits, regardless of whether the bug-fixing commit contains deleted lines.**

***Index Terms*—SZZ algorithm, data flow analysis, control flow analysis.**

## I. INTRODUCTION

MODERN software development evolves through a series of commits. When developers make commits, they save changes made to a set of files within a repository. Each commit captures the state of the repository as a snapshot at a specific moment. To facilitate the management of commits, version control systems such as Git have been developed. Among various types of commits, bug-inducing commits [1] hold particular significance, as they provide essential insights into the origins and evolution of bugs. Consequently, researchers have devoted additional attention to them. Prior studies have been conducted to understand the characteristics of bug-inducing commits [2], [3], [4], [5] and introduced numerous techniques to identify them [6], [7], [8], [9]. Building upon these identified bug-inducing commits, researchers have also proposed methods for Just-in-Time defect detection [9], [10], [11], [12] and automatic program repair [13], [14]. They have also worked on analyzing factors for software quality [15], [16], and pinpointing affected software versions of a vulnerability [17].

Currently, the primary approaches for identifying bug-inducing commits are the original SZZ algorithm [6] and its variants. The original SZZ algorithm was introduced by Sliwerski, Zimmermann, and Zeller in 2005 [6]. They assume that the deleted lines in the bug-fixing commit introduce the bug. Thus, the algorithm operates by tracing back the deleted lines within bug-fixing commits. This tracing process aims to identify the commits that last modified those deleted lines. These identified commits are then marked as bug-inducing commits. Although the original SZZ algorithm has achieved great success, it suffers from low precision because bug-fixing commits may contain noise (e.g., blank lines, comments, and meta-changes) that is unrelated to bug fixes. To address this issue, numerous variants have been introduced [7], [8], [9]. Typically, these variants make use of static methods to filter out noise in bug-fixing commits. Additionally, deep learning methods have also been explored to further improve precision [18].

However, previous research shows that the abovementioned methods fail to address the challenge posed by bug-fixing commits with only added lines [19]. When it happens, tools like *git blame* or *annotate* cannot identify deleted lines to link them with bug-inducing commits. Recent research has highlighted the prevalence of these kinds of commits in datasets, which significantly impacts the recall of SZZ algorithms. For instance, a dataset collected by Rezk et al. [19], containing 12 Apache projects, revealed that the highest percentage of such instances reached 11.72%. Further investigations, such as the study conducted by Lyu et al. [20], have demonstrated even higher occurrences of bug-fixing commits only containing added lines. Lyu et al.'s dataset reported that 17.46% of bug-fixing commits

Received 7 April 2024; revised 18 September 2024; accepted 22 September 2024. Date of publication 9 October 2024; date of current version 14 November 2024. This research was supported in part by the National Key Research and Development Program of China under Grant 2021YFB2701102, in part by the National Science Foundation of China under Grant 62372398, Grant 62302447, Grant 62202419, Grant 72342025, and Grant U20A20173, in part by the Fundamental Research Funds for the Central Universities under Grant 226-2022-00064, and in part by Zhejiang Provincial Natural Science Foundation of China under Grant LY24F020008. Recommended for acceptance by C. Treude *(Corresponding author: Lingfeng Bao.)*

Lingxiao Tang and Lingfeng Bao are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310007, China, and also with Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute, Hangzhou 310052, China (e-mail: 12421037@zju.edu.cn; lingfengbao@zju.edu.cn).

Chao Ni is with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310007, China (e-mail: chaoni@zju.edu.cn).

Qiao Huang is with the School of Computer Science and Technology, Zhejiang Gongshang University, Hangzhou 310018, China (e-mail: qiaohuang@zjgsu.edu.cn).

Digital Object Identifier 10.1109/TSE.2024.3468296

are solely comprised of added lines, indicating that these kinds of commits are more prevalent than previous findings.

Despite the prevalence of such commits and their negative impact on SZZ algorithms, few methods have been proposed to mitigate this issue, let alone evaluating them with a developer-informed oracle. To the best of our knowledge, the A-SZZ algorithm, proposed by Sahal et al. [21], is the main method intended to address this challenge. This algorithm operates by first identifying added lines, then locating the block encompassing these additions, and finally tracing back all lines within the block. However, our experimental results reveal that this algorithm suffers from significantly low precision, rendering it impractical for real-world use. Researchers [19] have proposed enhancements to the A-SZZ algorithm. However, these improvements are constrained to specific scenarios, such as method override, logging operations, and class expansions, limiting their applicability. Furthermore, they have not been evaluated under a developer-informed oracle and their evaluation lacks reliability.

In this paper, we present a novel approach called SEM-SZZ to tackle this issue. We conducted a study to explore the position of unmodified lines that could be traced back to identify bug-inducing commits. The results of this study revealed that in most cases if a bug-inducing commit could be found by tracing back unmodified lines within a function, it could typically be identified by tracing back only the two unmodified lines nearest the added lines, rather than tracing back the entire block encapsulating the added lines.

Building upon these findings, we further refined our approach by incorporating techniques from previous studies [7], [8] to filter out noise in the unmodified lines, such as blank lines, comments, and meta-changes. Despite these refinements that resulted in significantly higher precision compared to the A-SZZ algorithm, they still suffered from low precision.

To further enhance precision, we conduct a more fine-grained analysis. We begin by performing static program slicing to extract program parts relevant to the bug. Then, we collect and compare the states of the two versions of the program. One version is the most recent commit previous to the bug-fixing commit, while the other is the bug-fixing commit itself. This process incorporates data flow analysis and control flow analysis. Based on this analysis, we aim to identify which set of statements in the previous version contributed to the bug. Subsequently, when locating the bug-inducing commit, we traced back from the bug-fixing commit to identify the earliest commit that contains all these buggy statements, designating it as the bug-inducing commit.

Compared to previous methods, SEM-SZZ has two advantages. Firstly, it improves precision by more fine-grained analysis. Secondly, our method can be applied to all scenarios. Note that we can also adopt our approach to bug-fixing commits with deleted lines since the core idea of our approach is to compare the states of the two versions of the program and identify the statements contributing to the bug. Thus, based on the core idea, we can apply SEM-SZZ in all scenarios, regardless of whether they contain deleted lines.

To evaluate our approach, we utilize the dataset proposed by Lyu et al. [20]. Among all available datasets, we select this one for two primary reasons. Firstly, the dataset is annotated by the project's developers, ensuring its accuracy and reliability. Secondly, it contains the largest number of bug-fixing commits with only added lines compared to other datasets. This dataset, built on the Linux kernel, contains over ten thousand bug-fixing commits without deleted lines. Thus, it is much more reliable than previous datasets, which are manually annotated and small-scale. We assess our approach by attempting to address the following questions:

**RQ1: How effective is our approach in identifying bug-inducing commits from bug-fixing commits with only added lines?**

In this research question (RQ), we first compare SEM-SZZ with our proposed baselines and A-SZZ across all bug-fixing commits with only added lines. We also notice that some bug-inducing commits cannot be found even by tracing back all lines in the function and some bug-fixing commits do not contain changed lines within functions. All methods mentioned above cannot identify the first type of bug-inducing commits and SEM-SZZ is not designed to handle the second type of bug-fix commits. Thus, we remove these redundant test cases and then evaluate all methods. The results demonstrate that our proposed baselines significantly outperform A-SZZ, and SEM-SZZ surpasses all baselines significantly, with an improvement from 17% to 19% in F1-score.

**RQ2: How effective is SEM-SZZ in identifying bug-inducing commits from bug-fixing commits with deleted lines?**

In this research question (RQ), we compare our method with baselines to identify bug-inducing commits from bug-fixing commits with deleted lines. Our goal is to determine whether SEM-SZZ can effectively handle bug-fixing commits with deleted lines. The result demonstrates that SEM-SZZ enhances precision by 7% and F1-score by 7%, respectively.

**RQ3: How effective are the key components of SEM-SZZ?**

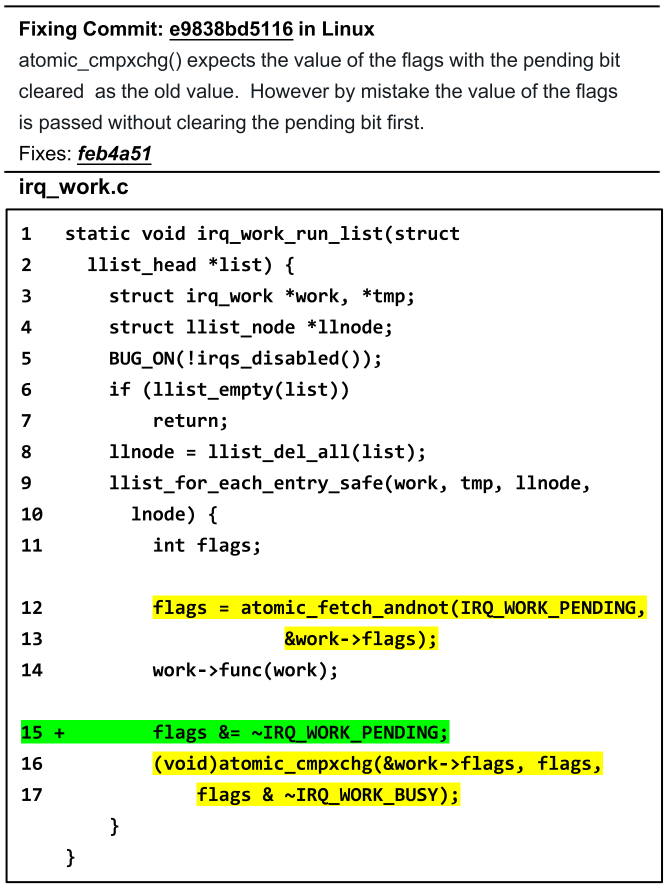
In this research question (RQ), we conduct an ablation experiment to verify the effectiveness of each key component. The result shows that each key design, including data flow comparison, path constraint comparison, the method to locate the bug-inducing commit and the line similarity threshold, contributes to the overall performance.

**RQ4: How effective is SEM-SZZ in terms of time cost?**

An approach that consumes too much time is impractical for deployment in real-world scenarios. In this RQ, we want to investigate the time efficiency of our approach. The result shows that our approach takes approximately 1.95 seconds to handle a fixing commit from scratch, which can be deployed in practice.

In summary, our contributions can be summarized as follows:

- We provide a new insight on how to find bug-inducing commits based on bug-fixing commits with only added lines. Our study reveals that most bug-inducing commits can be identified by tracing back nearby unmodified lines of the added lines.
- We introduce a novel approach for pointing out statements contributing to the bug from bug-fixing commits, which compares the program's state between two versions. This approach can be applied to all scenarios, regardless of whether the bug-fixing commit contains added lines.



**Fixing Commit: e9838bd5116 in Linux**
atomic_cmpxchg() expects the value of the flags with the pending bit cleared as the old value. However by mistake the value of the flags is passed without clearing the pending bit first.
Fixes: ***feb4a51***

**irq_work.c**

```
1   static void irq_work_run_list(struct
2     llist_head *list) {
3       struct irq_work *work, *tmp;
4       struct llist_node *llnode;
5       BUG_ON(!irqs_disabled());
6       if (llist_empty(list))
7           return;
8       llnode = llist_del_all(list);
9       llist_for_each_entry_safe(work, tmp, llnode,
10        lnode) {
11          int flags;

12          flags = atomic_fetch_andnot(IRQ_WORK_PENDING,
13                      &work->flags);
14          work->func(work);

15 +        flags &= ~IRQ_WORK_PENDING;
16          (void)atomic_cmpxchg(&work->flags, flags,
17              flags & ~IRQ_WORK_BUSY);
        }
    }
```

Fig. 1. A motivation example.

- We introduce a new method for locating bug-inducing commits by identifying the earliest commit that introduces all these buggy statements and designating it as the bug-inducing commit.
- Our experimental results demonstrate that SEM-SZZ outperforms all other baselines in all scenarios.

## II. MOTIVATION

In this section, we present a motivation example to illustrate the need for an alternative approach.

Fig. 1 illustrates our motivation example, showing a bug-fixing commit in the Linux kernel [22]. Due to space constraints, only the most significant part of the patch is displayed. The commit message clearly indicates the reason behind the evolution of the bug. Specifically, the previous version of the program fails to clear the pending bit of the variable $flags$ before passing it to the function $atomic\_cmpxchg$. The developer addresses this issue by introducing a clear-bit operation through an add-and-assign operation at line 15.

The previous approach, the A-SZZ algorithm [21], identifies the block that encompasses the added lines. In this example, it covers lines ranging from line 11 to 17. While it successfully identifies the bug-inducing commit, it also introduces significant noise. For instance, we can easily observe that the variable definition statement at line 11 is unrelated to the bug. The problem worsens when the block contains numerous lines. Most of these lines are unrelated to the bug.

From the motivation example, we can make the following observations:

**Observation 1. Most of the bug-inducing commits can be found by tracing back the unmodified lines near the added lines.** For instance, in this example, the bug-inducing commit can be located by tracing back to line 13, which is only two lines away from the added lines. Therefore, there is no need to trace back the entire block. In fact, subsequent experiments reveal that by tracing back only the two nearest unmodified lines adjacent to the added lines, we can identify 85% of bug-inducing commits with a much higher precision than the A-SZZ algorithm.

**Observation 2. To improve precision, we need to perform a more fine-grained analysis.** Although tracing back the unmodified lines near the added lines is effective in locating bug-inducing commits, it still suffers from low precision. For example, line 14 is unrelated to the variable $flags$ and therefore unrelated to the bug. A more fine-grained analysis can resolve this issue. By conducting a data flow analysis on two versions of the program, one for the buggy version and one for the fixed version, we can observe that the sole difference lies in the data flow of the $flags$ variable. In the previous version, the data flow of $flags$ extends from line 12 to line 16. In the current version, it flows from lines 12, and 15 to 16, with a new add-and-assign operation at line 15. Thus, we can exclude the irrelevant line 14.

**Observation 3. The bug arises from the combined impact of multiple statements, each of which is indispensable for its occurrence.** From the commit message, it is obvious that the bug requires two specific conditions. First, the variable $flags$ must be assigned without clearing the pending bit (lines 12-13). Secondly, the variable $flag$ must be passed to the function $atomic\_cmpxchg$ (lines 16-17). It's the interaction between these statements that triggers the bug. Thus, when pinpointing bug-inducing commits, it's essential to ensure that both statements are present within the identified commit, rather than just one of them.

## III. APPROACH

Building on the motivation example, our proposed method aims to effectively identify all statements contributing to the appearance of the bug. The architecture of SEM-SZZ is shown in Fig. 2.

Observation 1 leads our approach to focusing on program segments near the added lines. Thus, we begin by performing program slicing [23], [24] to extract relevant program parts close to the additions. Based on Observation 2, after slicing the program, we delve into a more detailed analysis. This involves comparing the states of both versions of the program, one with the added lines and one without. Through thorough comparison and analysis, we identify specific statements associated with the added lines, pinpointing them as the triggers for the bug. Drawing on Observation 3, we proceed to locate bug-inducing commits by finding the earliest commit that contains all buggy statements.

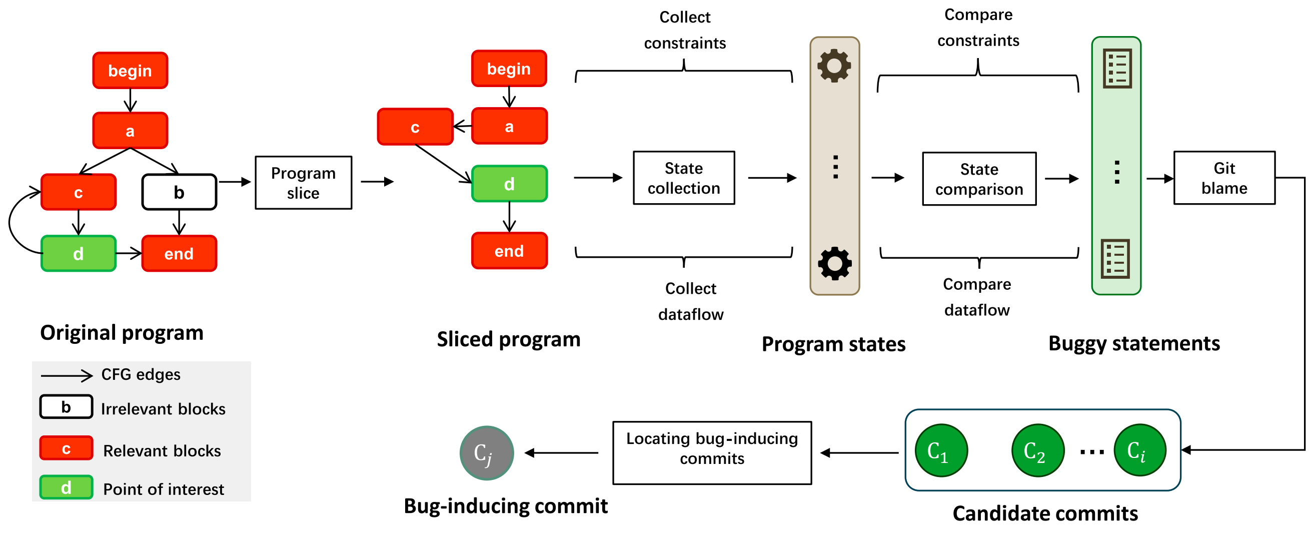


Fig. 2. Overview of SEM-SZZ.

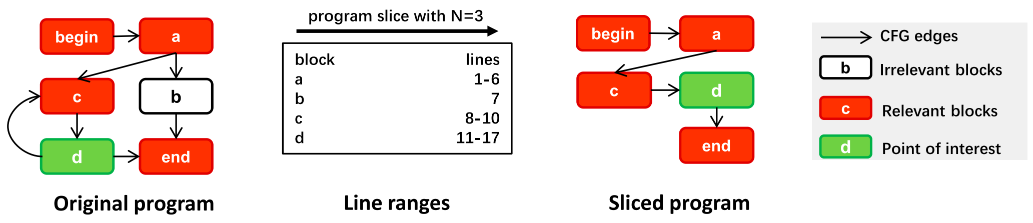


Fig. 3. An example of program slice.

## A. Program Slicing

In this section, we present the program slicing process. The goal of this process is to pick out specific parts of the program that are relevant to the bug.

Firstly, we extract the function from the fixed version containing the added lines. Then, we adapt the approach similar to the previous research [25] for an intraprocedural program slicing. During the program slicing process, we initially identify the point of interest, which in this case is the basic block containing added lines. Subsequently, we conduct both forward and backward program slicing centered around this point. In this way, we can avoid the noise introduced by the irrelevant lines.

In detail, we initially construct the control flow graph [26] for the function, splitting the function into a set of basic blocks. Then we pick out the basic block containing the added line as the point of interest. Finally, we use the depth-first-search algorithm to search its $N$ nearest predecessors and successors. Here, the parameter $N$ is the maximum search step, indicating the search scope of our program. When the search step reaches $N$, we save all the basic blocks encountered during the searching process and exclude all other parts of the program.

Here, our slice criterion is based on basic blocks with added code statements instead of variables. We do this for two reasons. Firstly, there may be many changed statements in a function, resulting in many changed variables. Handling them one by one is time-consuming. By dealing with changed basic blocks, we can address many code statements and variables once at a time. Secondly, slicing based on variables may introduce lines far from the added lines, which can introduce noise. Instead, by slicing only the basic blocks near the changed basic blocks, we can avoid this issue.

Fig. 3 illustrates an example of program slicing, based on the function depicted in the motivation example. As depicted in the figure, the original function is divided into six basic blocks, with line ranges also indicated. The added line number is 15. Thus, the basic block `d` contains the added line and we designate it as the interesting point. Here we set N to 3. We begin to search its three nearest predecessors (in this case, blocks `c`, `a` and `begin`). Subsequently, we explore its three nearest successors. Note that each block is visited only once throughout the process. Thus, we identify the `end` block as the only successor. As a result, our slicing result contains basic blocks `begin`, `a`, `c`, `d`, and `end`.

## B. State Collection

Having isolated the part of the program relevant to the bug, our next step is to collect the states of the program within this portion. We borrow the concept program's state from symbolic execution [27], [28]. In symbolic execution, the execution engine maintains the program's states $(\sigma, \pi)$ for each execution path, where $\sigma$ represents a table that maps each variable to its corresponding symbolic value $\alpha_i$, and $pi$ denotes the

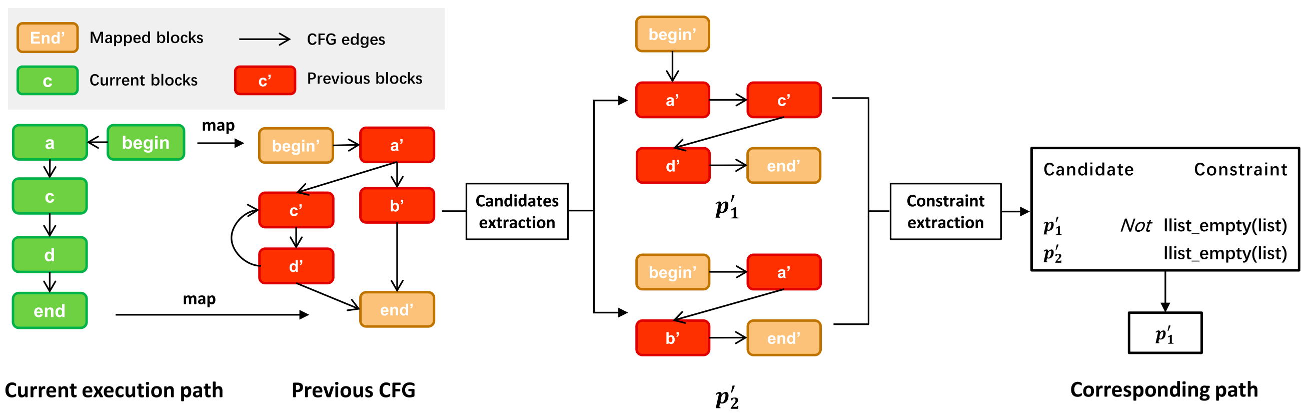


Fig. 4. Candidate selection.

```
Algorithm 1: obtaining the program's state after executing the path.
   Input  : path, the input path
   Output: state, the program's state for the execution
           path.
1  for statement in path do
2      if statement.type in {while, if, switch, for} then
3          state.constraints.add(statement)
4      if statement.type in {expression} then
5          variables ⟵ extractVars(statement)
6          for variable in variables do
7              state.variable.dataflow.append(statement)
8  return state
```

| Constraint | Not llist_empty(list) |
|---|---|
| variable | dataflow |
| list | 6, 8 |
| llnode | 4, 8, 9 |
| work | 3, 9, 13, 14, 16 |
| tmp | 3, 9 |
| flags | 11-12, 15, 16-17 |

Fig. 5. A program state example.

path constraints, signifying all branch conditions taken to reach each statement in the path. Given that our goal is to discern the differences between the states of the two versions of the program—the bug-fixed version and the buggy version, there is no imperative need for the symbolic value of each variable. Instead, we collect each variable's data flow for simplicity.

As shown above, in order to collect the program's state, we first construct its execution paths. We start from the basic blocks without predecessors and then traverse recursively through all their successors. When we reach basic blocks with no successors, we mark all statements encountered in sequential order as an execution path. For example, in the sliced program illustrated in Fig. 3, we start from the `begin` block and traverse its successors, including blocks `a`, `c`, `d`, until we arrive at the `end` block, which has no further successors. During the process, we add each statement encountered to the execution path.

Following path extraction, we proceed to generate the program's state for the execution path. Algorithm 1 outlines the process. We iterate through the statements in the execution path. If the statement type belongs to path constraints (line 2), we include the statement in the path constraints. Otherwise, we extract the variables from the statement and incorporate them into the data flow of these variables (lines 4-7).

Fig. 5 illustrates the program's state for the previously collected execution path in the motivation example. Upon encountering the statement at line 3, we extract the variables `work` and `tmp` from the statement. Consequently, we add line number 3 to their data flow, as depicted in the figure. Additionally, upon reaching line 6, we easily identify it as an if statement related to path constraint. Since our path executes from block `c` to `d`, and it does not satisfy the condition, we negate the condition and store it in the path constraint, as shown in Fig. 5.

## C. State Comparison

After collecting the states, our subsequent step involves comparing the states of the execution paths from the current version with those from the previous version. This comparison enables us to identify discrepancies in their states, ultimately helping us determine the unmodified statements contributing to the bug.

For each execution path in the current version, our objective is to discover how the added lines affect the path, making its state different from the previous version. To enable the comparison, the first step involves identifying the corresponding execution path in the previous version and obtaining its state.

Our goal is to find an execution path in the previous version that most closely resembles the one in the current version. Thus, we can capture the differences in their states more precisely. To get such a corresponding path, we begin by mapping the first and last basic blocks in the current execution path to their counterparts in the previous version of the program. Subsequently, we traverse from the first block in the previous version,

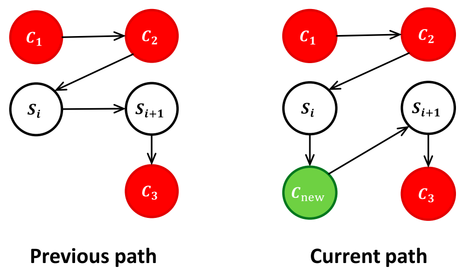


Fig. 6. Constraint comparison example.

endeavoring to find a path to the last block. It is important to note that we may encounter multiple candidates during this step. For each candidate in the previous version, we initially prioritize the one that shares the highest number of constraints with the current execution path. If there are multiple candidates, preference is given to the one with the most common lines.

To illustrate this, we use the extracted execution path $p$ from the motivation example. To derive its corresponding path in the previous version, we initially map its first and last basic blocks to their counterparts in the previous version. These correspond to the `begin'` block and the `end'` block, respectively, as illustrated in orange in Fig. 4.

Next, we attempt to find out paths from the first mapped block to the last mapped block in the previous version. In this case, we aim to find paths from the `begin'` block to the `end'` block in the previous version of the program. We use the depth-first-search algorithm to find out candidates. As depicted in the figure, we encounter two candidates: $p'_1$ and $p'_2$. After comparing their constraints, we observe that candidate $p'_1$ shares more constraints with $p$, leading us to designate it as the corresponding path.

After path matching, each path $p$ with the state $s$ in the current version has been paired with a corresponding path $p'$ with the state $s'$ from the previous version. Now, we initially compare the two states and proceed to find out which unmodified statements in the previous version contribute to the occurrence of the bug.

The detailed algorithm is outlined in Algorithm 2. Initially, we compare the constraints in two program states. If the constraints in the previous path are not identical to those in the current path, we first identify the index i where the constraint $preCons[i]$ in the previous version is different from the constraint $curCons[i]$ in the current version(lines 1-3). In case of a conflict condition, we assume that the previous version of the program may neglect the condition $curCons[i]$ from the statement $S_i$ to the statement $S_{i+1}$, where $S_i$ is the last statement before $curCons[i]$ and $S_{i+1}$ is the first statement after $curCons[i]$. Consequently, we utilize the function $extractStmts$ to extract $S_i$ and $S_{i+1}$, subsequently mapping them to the previous version and adding them to the buggy statements (lines 4-5).

We provide a constraint comparison example in Fig. 6. In the previous version, the path executes from $S_i$ to $S_{i+1}$ without a condition check while in the current version, it needs to satisfy $C_{new}$. The newly added constraint is highlighted in green while the original constraints are highlighted in red. Here the constraints in the previous version are $C_1$, $C_2$, and $C_3$. The current version of the program has constraints $C_1$, $C_2$, $C_{new}$ and $C_3$. We compare all constraints in sequential order and find that $C_3$ is conflicted with $C_{new}$ at index 2. Thus, we extract the last statement $S_i$ before $C_{new}$ and the first statement $S_{i+1}$ after $C_{new}$. We map these two statements to the previous version and add them to buggy statements.

**Algorithm 2:** obtain the buggy statements.

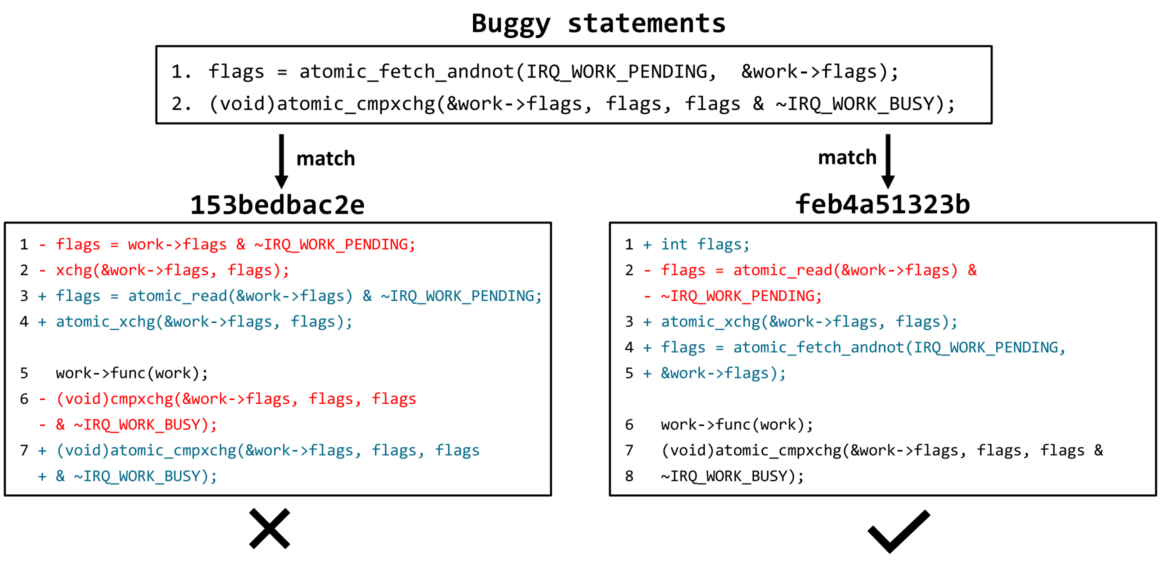
```
   Input  : $s$, state of the current path.
            $s'$, state of the previous path.
   Output: $bStmts$, the obtained buggy statements.
   $i \longleftarrow 0$
   $preCons \longleftarrow s'.constraints$
   $curCons \longleftarrow s.constraints$
1  for $i < curCons.size()$ and $i < preCons.size()$ do
2      if $curCons[i] \neq preCons[i]$ then
3          break
       $i \longleftarrow i + 1$
4  if $i \neq curCons.size()$ then
5      $bStmts$.add(extractStmts ($curCons$ [i],$s$))
6  for $var$ in $s'.vars$ do
       $i \longleftarrow 0$
7      $preData \longleftarrow s'.var.data$
8      $curData \longleftarrow s.var.data$
9      for $i < curData.size()$ and $i < preData.size()$
        do
10         if $curData[i] \neq preData[i]$ then
11             break
           $i \longleftarrow i + 1$
12     if $i \neq curData.size()$ then
13         $bStmts$.add($preData$ [i-1],$preData$ [i])
```

For each variable in the previous path, we compare its data flow with that in the current path (lines 10-11). Again, we first locate the conflict index. Then, we add the conflicting index and its preceding index's corresponding statements to the buggy statements (lines 12-13). Note that a variable's data flow is the sequence of code statements that define, assign, modify, or use the variable in the path. Additionally, when we compare the data flow of a variable, we do not consider the control flow.

In the case of the motivation example, we first examine the constraints and identify no differences. Subsequently, we analyze the data flow of each variable. Upon inspection, we observe that the data flow of the variable `flags` in the previous version encompasses two statements 11-12, 16-17, while in the current version, it comprises three statements 11-12, 15, 16-17. Consequently, a conflict arises at index 1, as we illustrate in green in Fig. 5. As a result, we include the statements 11-12, 16-17 as the statements contributing to the bug.

## D. Locating Bug-Inducing Commits

After identifying all buggy statements, the subsequent step involves tracing back the commit history to pinpoint the bug-inducing commit. We adopt an approach similar to the "per-



```
1. flags = atomic_fetch_andnot(IRQ_WORK_PENDING,  &work->flags);
2. (void)atomic_cmpxchg(&work->flags, flags, flags & ~IRQ_WORK_BUSY);
```

```
1 - flags = work->flags & ~IRQ_WORK_PENDING;
2 - xchg(&work->flags, flags);
3 + flags = atomic_read(&work->flags) & ~IRQ_WORK_PENDING;
4 + atomic_xchg(&work->flags, flags);

5   work->func(work);
6 - (void)cmpxchg(&work->flags, flags, flags
  - & ~IRQ_WORK_BUSY);
7 + (void)atomic_cmpxchg(&work->flags, flags, flags
  + & ~IRQ_WORK_BUSY);
```

```
1 + int flags;
2 - flags = atomic_read(&work->flags) &
  - ~IRQ_WORK_PENDING;
3 + atomic_xchg(&work->flags, flags);
4 + flags = atomic_fetch_andnot(IRQ_WORK_PENDING,
5 + &work->flags);

6   work->func(work);
7   (void)atomic_cmpxchg(&work->flags, flags, flags &
8   ~IRQ_WORK_BUSY);
```

Fig. 7. Locating bug-inducing commits.

fect test" model proposed by Rodríguez-Pérez et al. [29]. The model consists of two parts. The first part involves determining whether a commit is buggy. They assume there exists a "perfect test" to accomplish this. If a commit can pass the test, it is considered correct; otherwise, it is deemed buggy. The second part involves locating the bug-inducing commit. They trace back from the bug-fixing commit until encountering the earliest commit that fails the "perfect test," marking it as the bug-inducing commit. However, in practice, there is always no applicable "perfect test" available. Therefore, they replace it with manually designed tests based on bug reports.

Based on the model mentioned above, our method to locate the bug-inducing commits also consists of two parts. To determine whether a commit is buggy, we replace the "perfect test" with the buggy statements collected above. As we mentioned above, each buggy statement is indispensable to the occurrence of the bug. Thus, we consider the commit as buggy only when it contains all buggy statements or it contains statements with the same semantic meaning as all buggy statements. To determine whether two statements have the same semantic meaning, we leverage the methodology proposed by V-SZZ [17]. If two statements have a line similarity beyond a threshold, we believe that they have the same semantic meaning. Here, we also use the same method to compute line similarity and set the similarity threshold to 0.75 as V-SZZ.

Suppose that we have two strings $S_1$ and $S_2$. We first calculate the sum of the length of $S_1$ and $S_2$ as in formula 1. Then we calculate the line similarity shown in formula 2, where the $levenshteinDistance$ function is used to calculate the Levenshtein edit distance [30] of the two strings. Line similarity ranges from 0 to 1. Higher values indicate a greater likelihood that two code statements share the same semantic meaning.

$$L = len(S_1) + len(S_2) \tag{1}$$

$$S = (L - levenshteinDistance(S_1, S_2))/L \tag{2}$$

To locate the bug-inducing commits, we trace back all buggy statements, generating a set of candidate commits. Subsequently, we scrutinize these candidates to identify the earliest buggy commit and designate it as the bug-inducing commit, which is the same as the model above.

Fig. 7 illustrates the process of locating bug-inducing commits. After obtaining statements contributing to the bug, we first use the `git blame` tool on the statements and obtain two candidates: commit 153bedbac2e and commit feb4a51323b. Then we begin to check which commit first introduces all buggy statements. We first check the earliest commit 153bedbac2e and find that no statement has a line similarity over 0.75 with the statement `flags = atomic_fetch_andnot(IRQ_WORK_PENDING, & work-> flags)`. The most closely related statement is line 3, but its similarity is still under 0.75. Thus, 153bedbac2e is not the bug-inducing commit. Then, we check commit feb4a51323b and find that it contains all statements in the buggy statements. Thus, we designate it as the final bug-inducing commit, which is the same as the developers' message in Fig. 1.

## *E. Extending SEM-SZZ to Bug-Fixing Commits With Deleted Lines*

While our method is primarily designed for bug-fixing commits with only added lines, the approach of comparing program states and identifying buggy statements can also be applied to commits with deleted lines. However, some adjustments are necessary for this scenario, particularly with respect to deleted lines.

First, during program slicing, we mark the basic block containing deleted lines as the point of interest and search its nearest N predecessors and successors. Second, when comparing program states, we focus more on deleted lines. Specifically, if we find differences in path constraints between the previous and current versions, we identify the deleted lines related to those

**Fixing Commit: 83ab7dad06b in Linux**
rtc: pcf2123: Add missing error code assignment before test.

```
1  static ssize_t pcf2123_store(struct device *dev, struct
2  device_attribute *attr,
3  -     const char *buffer, size_t count) {
4  +     const char *buffer, size_t count)
5  {
6          struct pcf2123_sysfs_reg *r;
7          unsigned long reg;
8          unsigned long val;
9          int ret;
10         r = container_of(attr, struct pcf2123_sysfs_reg, attr);
11         ret = kstrtoul(r->name, 16, &reg);
12         if (ret)
13             return ret;
14         ret = kstrtoul(buffer, 10, &val);
15         if (ret)
16             return ret;
17 -       pcf2123_write_reg(dev, reg, val);
18 +       ret = pcf2123_write_reg(dev, reg, val);
19         if (ret < 0)
20             return -EIO;
21         return count;
22 }
```

Fig. 8. An example of handling bug-fixing commits with deleted lines.

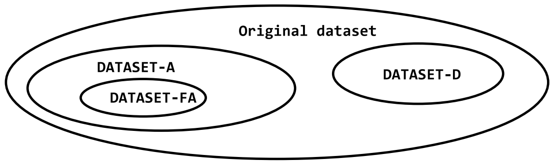


Fig. 9. Datasets relationship.

**Fixing Commit: b0138364da1 in Linux**
module_param_named() requires linux/moduleparam.h

```
  * WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
  SOFTWARE.
  */
+ #include <linux/moduleparam.h>
  #include <drm/ttm/ttm_execbuf_util.h>
  #include "virtgpu_drv.h"
```

Fig. 10. A bug-fixing commit involving no changes within functions.

constraints and mark them as buggy statements. Similarly, when comparing the data flow of each variable, if there's a difference between versions, we identify and mark the related deleted lines as buggy statements.

Fig. 8 illustrates why comparing paths is necessary. If we blamed all deleted lines, lines 3 and 17 would be included. However, line 3 is simply replaced by line 4 and does not affect the program's state. By comparing paths, we can see that the data flow of variables `buffer` and `count` is not affected, allowing us to exclude line 3 and reduce noise.

## IV. Experiment Setup

### A. Experiment Setting

The experiment was conducted on a server equipped with an Intel(R) Xeon(R) Gold 6226R CPU, running the Ubuntu operating system. We utilized `gitpython` to extract source files from both the previous and current versions of the program. The control flow graph was constructed based on the AST generated by the tree-sitter parser [31]. Following the V-SZZ algorithm, we set the line similarity threshold as 0.75. Notably, we set the maximum search step parameter N to three. Further details on the impact of various maximum search steps will be discussed in the subsequent section. Moreover, we implement all baselines based on tools provided by Lenarduzzi et al. [32].

### B. Data Preparation

To evaluate our method, we require a high-quality dataset. While several researchers [33], [34] have introduced datasets containing bug-inducing and bug-fixing commits, a common challenge is the lack of annotations by project developers, potentially leading to inaccuracies. Recently, some datasets [35], [36] with developers' annotations have been proposed, but all of them have very limited examples of bug-fixing commits with no deleted lines. For example, after filtering, the dataset proposed by Rosa et al. [35] contains only about 20 bug-fixing commits only with added lines. Considering these factors, we opted to utilize the dataset proposed by Lyu et al. [20], which is constructed based on the Linux kernel and contains 79,649 bug-fixing commits. There are several reasons for this choice. Firstly, it is annotated by developers, ensuring its accuracy. The developers leave the link to the bug-inducing commit in the commit message of the bug-fixing commit. Secondly, it contains a significant number of bug-fixing commits without any deletion lines. To evaluate our method, we partition the original dataset into multiple datasets, as illustrated in Fig. 9.

We initially pick out bug-fixing commits containing c files with only added lines from the original dataset, yielding 11,426 commits, which we call DATASET-A. Here, the suffix "A" means added lines. To better demonstrate the effectiveness of Sem-SZZ, we filter out two types of commits from DATASET-A. The first type of commits does not involve changes within functions. These commits often only involve changes to header file includes, struct definitions, or global variable declarations. We have provided an example in Fig. 10. The commit fixes the bug only by adding a new header file. The second type involves changes within functions, but even by tracing all lines in the changed functions, we still cannot identify the bug-inducing commits. Our approach is not designed for the first type, and no approaches, including baselines, can handle the second type. Thus, we filter out these two types of commits from DATASET-A and get DATASET-FA, which contains 6,233 bug-fixing commits. Here, the suffix "FA" stands for the filtered bug-fixing commits with only added lines. Finally, to evaluate our method's effectiveness in identifying bug-inducing commits from bug-fixing commits with deleted lines, we extract bug-fixing commits containing c files with deleted lines and changes in functions, resulting in 48,736 test

TABLE I
THE STATISTICS OF THE BUGS AND CORRESPONDING BUG-FIXING COMMITS IN THREE DATASETS

| DATASET | #Bug-Fixing | #Bug-Inducing | #SMALL | #LARGE |
|---|---|---|---|---|
| DATASET-A | 6,233 | 6,419 | 7,959 | 3,467 |
| DATASET-FA | 11,426 | 11,829 | 4,394 | 1,839 |
| DATASET-D | 48,736 | 50,978 | 16,615 | 25,975 |

cases, forming the DATASET-D. Here, the suffix "D" means the deleted lines. Note that as long as a bug-fixing commit contains deleted lines, it can be included in DATASET-D. Thus, DATASET-D includes bug-fixing commits with only deleted lines and bug-fixing commits with both added and deleted lines.

Table I presents the statistics of our three datasets. We categorize the bug-fixing commits following previous research [17], [18]. Notice that if a bug-fixing commit contains fewer than five changed lines, we categorize it as a small bug-fixing commit; otherwise, we categorize it as large. From the table, we can see that all three datasets contain multiple large bug-fixing commits. This is helpful when we want to demonstrate that our approach can scale to large bug-fixing commits. Moreover, we can also see that one bug-fixing commit may correspond to multiple bug-inducing commits.

## C. Baselines

Few previous research studies have proposed methods to identify bug-inducing commits that involve only added lines. To enhance the evaluation of our approach, we introduce a new set of baselines of our own, complementing the existing approaches. All the following baselines have the prefix "A". Here, the "A" stands for "added lines," indicating that these baselines are designed to handle bug-fixing commits with only added lines.

- **A-SZZ.** A-SZZ, proposed by Sahal et al. [21], is an approach that initially identifies the added lines, then locates the block encapsulating the new code change, and finally utilizes the git blame tool to trace back the historical changes of the code block.
- **AB-SZZ.** Building on the concept of B-SZZ [6], we introduce an additional baseline named AB-SZZ. Similar to the B-SZZ algorithm, which utilizes the git blame tool on each deletion line, our variant also aims to find lines for each added line to trace back. In contrast to the relatively coarse scope of A-SZZ, which covers the whole code block, our approach only selects the two nearest unmodified lines above and below each added line. In this way, we can guarantee that each added line has unmodified lines to trace back.

Our study also adopts the concept of B-SZZ variants, such as AG-SZZ, MA-SZZ, R-SZZ and L-SZZ to introduce baselines, which is the same as three previous studies [17], [20], [37]. Note that we exclude the RA-SZZ algorithm [9], because there are no existing tools to identify refactors in the C programming language.

- **AAG-SZZ.** The original AG-SZZ algorithm, proposed by Kim et al. [7], enhances the B-SZZ algorithm by filtering out non-semantic lines, such as blank lines, comment lines, and format modifications. Additionally, they leverage the annotation graph to gather more information than the `annotate` command. Given that the git blame tool already leverages the annotation graph, the AAG-SZZ algorithm only needs to filter out non-semantic lines identified in the AB-SZZ algorithm.
- **AMA-SZZ.** The original MA-SZZ algorithm is proposed by Da Costa et al. [8]. They notice that the existence of meta-changes introduces noise in the identified bug-inducing commits. Meta-changes refer to changes that do not alter the semantics of code, such as branch changes, merge changes, and property file changes. To implement AMA-SZZ, we need to filter out meta-changes based on the AAG-SZZ algorithm.
- **AR-SZZ and AL-SZZ.** The original R-SZZ and L-SZZ algorithms were proposed by Davies et al. [33]. The R-SZZ algorithm selects the most recent commit from the AG-SZZ algorithm results, while the L-SZZ selects the commit that changed the most lines. In the AR-SZZ and AL-SZZ algorithms, we adopt the same selection strategy as the R-SZZ and L-SZZ algorithms to select the final commit from the results produced by the AAG-SZZ algorithm.

To evaluate our approach in scenarios where the bug-fixing commit contains deleted lines, we adopt the approaches B-SZZ, AG-SZZ, MA-SZZ, L-SZZ and R-SZZ as in previous studies [17], [20], [37].

## D. Metrics

Following previous research practices [17], [18], [20], we adopt three widely used metrics (i.e., Precision, Recall, and F1-score) to assess the performance of our approach and other baselines.

**Precision** Precision gauges the accuracy of positive predictions made by the SZZ algorithm. In this context, we define $I_T$ as the SZZ algorithm's correctly identified true bug-inducing commits and $I_F$ as the falsely identified bug-inducing commits. Therefore, precision can be calculated as follows:

$$P = \frac{I_T}{I_T + I_F} \tag{3}$$

**Recall** is a measure of the SZZ algorithm's ability of capturing true bug-inducing commits among all bug-inducing commits. Here $I_T$ is defined as the SZZ algorithm's correctly identified true bug-inducing commits and $M$ is the number of all bug-inducing commits. Thus, recall can be calculated as follows:

$$R = \frac{I_T}{M} \tag{4}$$

**F1-score** is the harmonic mean of precision and recall. It provides a balance between these two metrics, taking both false positives and false negatives into account. After obtaining $P$ as

TABLE II
THE PERFORMANCE COMPARISONS BETWEEN ALL METHODS FOR FINDING ALL BUG-INDUCING COMMITS

| Approach | Precision | Recall | F1-score |
|---|---|---|---|
| A-SZZ | 0.07 | **0.64** | 0.13 |
| AB-SZZ | 0.26 | 0.56 | 0.36 |
| AAG-SZZ | 0.28 | 0.52 | 0.36 |
| AMA-SZZ | 0.28 | 0.52 | 0.36 |
| AR-SZZ | 0.39 | 0.38 | 0.37 |
| AL-SZZ | 0.37 | 0.36 | 0.36 |
| SEM-SZZ * | **0.45** | 0.43 | **0.44** |

TABLE III
THE PERFORMANCE COMPARISONS BETWEEN ALL METHODS FOR FINDING FILTERED BUG-INDUCING COMMITS

| Approach | Precision | Recall | F1-score |
|---|---|---|---|
| A-SZZ | 0.11 | **0.96** | 0.20 |
| AB-SZZ | 0.42 | 0.84 | 0.56 |
| AAG-SZZ | 0.44 | 0.79 | 0.57 |
| AMA-SZZ | 0.44 | 0.79 | 0.57 |
| AR-SZZ | 0.58 | 0.56 | 0.57 |
| AL-SZZ | 0.57 | 0.55 | 0.56 |
| SEM-SZZ | **0.68** | 0.66 | **0.67** |

the precision and $R$ as the recall, the F1-score can be calculated as follows:

$$F1 = \frac{2PR}{P + R} \quad (5)$$

## V. EXPERIMENT

This section presents the experiment results of our research questions.

### A. *Effectiveness of Our Method in Identifying Bug-Inducing Commits From Bug-Fixing Commits With Only Added Lines*

In this research question, we focus on two experimental settings: all bug-inducing commits and filtered bug-inducing commits. In the first setting, we concentrate on locating bug-inducing commits in DATASET-A. In the second setting, we focus on locating bug-inducing commits in DATASET-FA.

In DATASET-A, there are 2,411 commits that involve no changes within functions, accounting for 21% of the total test cases. Thus, when handling test cases in DATASET-A, we first check whether the current commit contains added lines in functions. If it does, we use SEM-SZZ to produce the result. If it does not, we use the AR-SZZ algorithm. This approach gives us the experimental results shown in the row SEM-SZZ * in Table II.

Table II presents the results of our approach and baselines for identifying all bug-inducing commits. Notably, the precision of some baselines, especially the A-SZZ algorithm, is relatively low, with a precision of less than 0.1. The best baseline AR-SZZ has a precision equal to 0.39, which is still lower than SEM-SZZ *. Thus, our approach improves precision by 22% compared to the best baseline AR-SZZ.

As shown in Table II, A-SZZ performs the worst in F1-score due to a high number of false positives. Our proposed baselines outperform A-SZZ significantly. The best baselines AAG-SZZ and AMA-SZZ outperform A-SZZ by 200% in F1-score. This indicates the effectiveness of tracing back the neighboring unmodified lines of added lines. Furthermore, we observe that the AB-SZZ algorithm and its variants perform almost the same, resembling findings from prior research [20] where B-SZZ, AG-SZZ, MA-SZZ, L-SZZ and R-SZZ demonstrated similar performance. Notably, the idea of filtering out meta-changes appears to have no effect in this context. In Table II, SEM-SZZ * also demonstrates the best balance between precision and recall, improving F1-score by 19% compared to the best baseline AR-SZZ.

We then re-run all methods on DATASET-FA, and the results are documented in Table III. Table III demonstrates a significant improvement in the performance of all methods. However, some baselines continue to struggle with low precision. Notably, the precision of the A-SZZ algorithm is still quite low, with only a slight improvement from 0.07 to 0.11. After filtering out certain commits, the precisions of the AB-SZZ, AAG-SZZ and MA-SZZ algorithms remain below 0.50, despite an improvement of around 0.16. From the table, it's evident that SEM-SZZ continues to outperform all baselines in precision, outperforming the best baseline AR-SZZ by 17%.

From Table III, we can also see that the unmodified lines, which can be traced back to find bug-inducing commits, mostly lie near the added lines. This is evident from the recall of the baselines, where they find most of the bug-inducing commits successfully. This corresponds to the introduction and the motivation example, where we claim that most of the bug-inducing commits can be found by tracing back the unmodified lines near the added lines.

Compared to all baselines, SEM-SZZ also achieves the best balance between precision and recall. This is evident in the F1-score, where SEM-SZZ achieves an F1-score of 0.67. It outperforms the best baselines by 18%, respectively. Therefore, we believe that SEM-SZZ is more effective in identifying bug-inducing commits which can be found by tracing back all lines within the function.

**RQ-1:** SEM-SZZ *is more precise in identifying bug-inducing commits compared to all baselines, increasing precision by 22%. Furthermore,* SEM-SZZ *achieves a notable improvement of 19% in F1-score compared to the best baseline.*

### B. *Effectiveness of Our Method in Identifying Bug-Inducing Commits From Bug-Fixing Commits With Deleted Lines*

Table IV presents the results of SEM-SZZ and baselines on identifying bug-inducing commits from bug-fixing commits in DATASET-D. The results indicate that all baselines perform

TABLE IV
THE PERFORMANCE COMPARISONS FOR FINDING BUG-INDUCING COMMITS FROM BUG-FIXING COMMITS WITH DELETED LINES

| Approach | Precision | Recall | F1-score |
|---|---|---|---|
| B-SZZ | 0.42 | 0.71 | 0.53 |
| AG-SZZ | 0.48 | 0.65 | 0.55 |
| MA-SZZ | 0.48 | 0.65 | 0.55 |
| R-SZZ | 0.58 | 0.55 | 0.56 |
| L-SZZ | 0.56 | 0.52 | 0.54 |
| SEM-SZZ | **0.62** | 0.59 | **0.60** |

TABLE V
THE PERFORMANCE COMPARISONS IN ABLATION STUDY

| Similarity | Precision | Recall | F1-score |
|---|---|---|---|
| SEM-SZZ-c | **0.69** | 0.31 | 0.47 |
| SEM-SZZ-d | 0.67 | 0.36 | 0.42 |
| SEM-SZZ-l | 0.65 | 0.64 | 0.64 |
| SEM-SZZ | **0.68** | 0.66 | **0.67** |

similarly. The AG-SZZ, MA-SZZ, R-SZZ and L-SZZ algorithms improve the precision compared to B-SZZ, but their F1-scores are almost the same. Notably, the recall of B-SZZ is relatively high. This implies that if there are deleted lines, it is very likely that we can find the bug-inducing commits by tracing back the deleted lines. This explains why we need to modify the approach when dealing with bug-fixing commits with deleted lines and why we need to pay more attention to deleted lines when they exist.

Additionally, from the table, we can observe that SEM-SZZ also outperforms all baselines. It improves the precision by 7%, compared to the best baseline R-SZZ. Our method also demonstrates a notable improvement in F1-score, surpassing the best baseline by 7%.

**RQ-3:** SEM-SZZ *outperforms all baselines in identifying bug-inducing commits from bug-fixing commits with deleted lines. It enhances precision by 7% and F1-score by 7% compared to the best baseline.*

## C. Effectiveness of Key Designs in SEM-SZZ

In this research question, we aim to evaluate the effectiveness of the key components in our approach. SEM-SZZ incorporates two primary designs for identifying buggy statements and one for pinpointing bug-inducing commits.

The first design involves comparing path constraints between the previous and current versions of the program. Here, the previous version refers to the first commit preceding the bug-fixing commit, while the current version refers to the bug-fixing commit itself. The second design involves comparing the data flow of each variable in both versions. In the third design, we utilize line similarity comparison to identify the earliest commit that introduces the statements contributing to the bug, marking it as the bug-inducing commit.

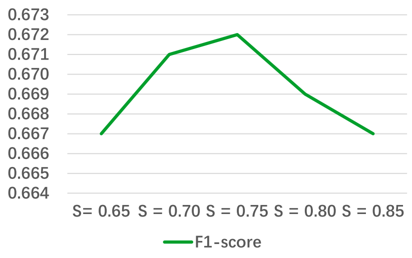


Fig. 11. The impact of line similarity threshold.

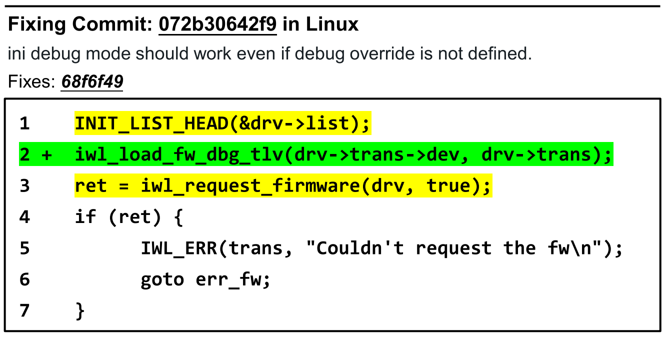
To do the ablation study, we first use DATASET-FA and compare SEM-SZZ with three of its variants: SEM-SZZ-c, SEM-SZZ-d, and SEM-SZZ-l, each missing one key component. Then we change the line similarity threshold and evaluate its impact on the SEM-SZZ's performance. In SEM-SZZ-c, we only compare the constraints and drop the comparison of data flow. Similarly, SEM-SZZ-d only focuses on comparing the data flow and drops the comparison of constraints. Both variants would cause fewer buggy statements to be identified. In SEM-SZZ-l, we do not use the line similarity comparison to identify bug-inducing commits, instead following V-SZZ's approach by selecting the earliest one.

From Table V, it's evident that SEM-SZZ achieves the highest F1-score compared to all of its variants. While SEM-SZZ-c demonstrates the best precision among all methods, dropping data flow comparison results in a significant decrease in recall. Similarly, dropping constraint comparison has a substantial negative impact on recall performance. Both constraint and data flow comparisons contribute almost equally to recall. Moreover, our method of locating bug-inducing commits enhances the method's ability to point out bug-inducing commits, improving all metrics. It improves precision by 5%, recall by 3%, and F1-score by 5% compared to SEM-SZZ-l. Fig. 11 shows that the choice of line similarity threshold also influences the performance. When the line similarity threshold rises from 0.65 to 0.75, the F1-score also increases. However, when the threshold increases from 0.75 to 0.85, the F1-score drops. This shows that 0.75 is an appropriate value for the line similarity threshold.

**RQ-2:** *Table V illustrates that each of our key designs contributes to the performance in identifying bug-inducing commits. Both constraint comparison and data flow comparison contribute equally to recall. Additionally, our method of locating bug-inducing commits enhances the performance across all metrics and 0.75 is the appropriate value for the line similarity threshold.*

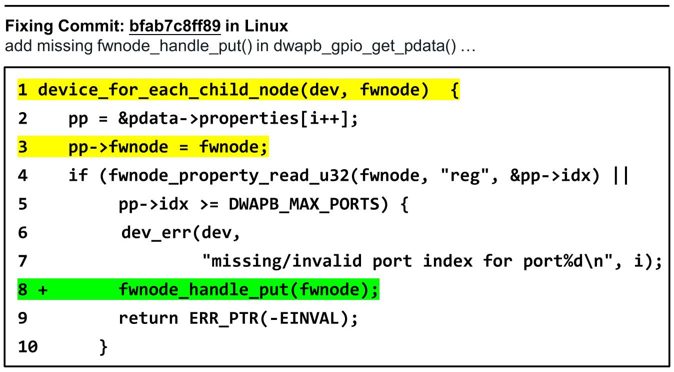
## D. Time Efficiency

In this section, our objective is to assess the time consumption of our approach during program analysis. We compare the performance of our SEM-SZZ with the baseline A-SZZ, both of which involve analyzing the program's structure.


**Fixing Commit: 072b30642f9 in Linux**
ini debug mode should work even if debug override is not defined.
Fixes: *68f6f49*

```
1     INIT_LIST_HEAD(&drv->list);
2 +   iwl_load_fw_dbg_tlv(drv->trans->dev, drv->trans);
3     ret = iwl_request_firmware(drv, true);
4     if (ret) {
5           IWL_ERR(trans, "Couldn't request the fw\n");
6           goto err_fw;
7     }
```

Fig. 12. A failing example.


**Fixing Commit: bfab7c8ff89 in Linux**
add missing fwnode_handle_put() in dwapb_gpio_get_pdata() …

```
1 device_for_each_child_node(dev, fwnode)  {
2    pp = &pdata->properties[i++];
3    pp->fwnode = fwnode;
4    if (fwnode_property_read_u32(fwnode, "reg", &pp->idx) ||
5         pp->idx >= DWAPB_MAX_PORTS) {
6         dev_err(dev,
7                 "missing/invalid port index for port%d\n", i);
8 +       fwnode_handle_put(fwnode);
9         return ERR_PTR(-EINVAL);
10     }
```

Fig. 13. An example to show the effectiveness of program slicing.

The outcome reveals that it takes approximately three hours and fifty minutes to process all examples in DATASET-A, equating to an average of 1.95 seconds per test case. Compared to A-SZZ, our approach takes approximately three times longer. This is attributed to the fact that our method involves analyzing both the previous and current versions of the program, whereas A-SZZ only needs to focus on the current version. Additionally, our approach requires collecting states and comparing them between two program versions. Despite the increased time overhead, we believe our approach remains practical and applicable in real-world scenarios.

**RQ-3:** SEM-SZZ *takes approximately 1.95 seconds to handle a fixing commit from scratch. This duration includes tasks such as extracting the source files, collecting and comparing states, and locating bug-inducing commits.*

## VI. Discussion

### A. Analysis for the Failure Case

In this section, we aim to analyze the reasons behind the failures observed in some cases. We randomly select 100 cases to analyze the reasons for the failures. After analysis, the failed reasons can be summarized as follows:

**Line similarity fails to determine whether a commit is buggy.** Out of the 100 cases analyzed, 28 cases fail because SEM-SZZ fails to determine whether a commit is buggy.

The main reason for this is the line similarity matching protocol used in our method. In our approach, for each statement contributing to the bug, if there all exists a line with string similarity beyond a threshold, we assume that the commit is influenced by the buggy. However, this method sometimes fails to accurately identify the bug-inducing commits. For instance, consider the commit 15273ffd7ef [38], which adds a new line `ret = 0;`. Our method generates two buggy statements: `if (!ret)` and `return ret;`. However, when locating the bug-inducing commit, the method encounters many statements like `if (ret)`, which exhibit high line similarity but have completely different semantic meanings. This can lead to incorrect judgments and result in finding false bug-inducing commits. This issue is particularly prevalent when the buggy statements contain many short statements.

**Failing to point out buggy statements correctly.** Another main reason for the failure is that our method fails to point out buggy statements correctly. This issue is responsible for most of the remaining failing cases. For instance, while the added line may influence the variable's data flow, analyzing the affected variable's data flow may not yield correct results.

Consider the commit 072b30642f9 [39], illustrated in Fig. 12. This commit introduces a new code line at line 2, highlighted in green, impacting the data flow of the variable `drv`. Our method, accordingly, compares its data flow with the previous version and identifies lines 1 and 3 as buggy statements. However, tracing back these unmodified lines does not lead us to the bug-inducing commit, resulting in the failure of our method in this scenario.

After carefully analyzing the bug-fixing commit and the bug-inducing commit, we find that the bug occurs because the bug-inducing commit attempts to introduce a new function in the program but forgets to use it. Specifically, it creates the `iwl_load_fw_dbg_tlv` function. However, the bug-inducing commit only defines the function and forgets to use it. Consequently, the bug-fixing commit fixes the bug by using the `iwl_load_fw_dbg_tlv` function in line 2. Therefore, we can conclude that the bug has nothing to do with the data flow of the `drv` variable. As a result, SEM-SZZ fails to identify the bug-inducing commit in this case.

### B. Effectiveness of Program Slicing in Finding Bug-Inducing Commits

Although tracing back the unmodified lines near the added lines is an effective approach for finding bug-inducing commits if we do not consider precision, there are still some bug-inducing commits that cannot be found in this way. In this section, we discuss whether program slicing, which explores N basic blocks near the added lines, helps find more bug-inducing commits that the baselines cannot detect.

In the experiment, we try to find bug-inducing commits that SEM-SZZ can detect but baselines can not. The experimental result shows that SEM-SZZ does find bug-inducing commits that the baselines cannot detect, accounting for 1.4% of the total test cases. One typical example is the commit bfab7c8ff89 [40], shown in Fig. 13. In this case, the bug occurs because

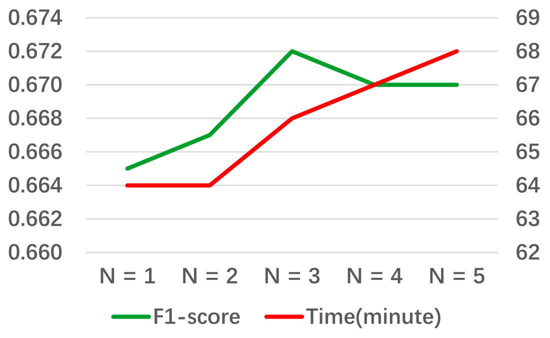


Fig. 14. The impact of parameter N.

the previous version of the program does not call the `fwnode_handle_put` function before returning. If we trace back the two nearest lines between the added line, we cannot find the bug-inducing commit. However, by comparing the differences in the data flow of the variable `fwnode`, we can locate lines 1 and 3. Tracing back from these lines allows us to successfully identify the bug-inducing commit.

## C. Impact of the Maximum Search Step

In this section, we aim to investigate the impact of the maximum search step parameter N. As mentioned earlier, given that most bug-inducing commits can be identified by tracing back nearby lines, we set the maximum search step parameter N to three. This parameter dictates that for each added line, we search three basic blocks above and below it.

In this section, we vary the parameter N from 1 to 5 to assess the performance of SEM-SZZ under different settings. Our evaluation focuses on key aspects such as F1-score and time efficiency. Here, to show the influence of the maximum search step on time efficiency more clearly, we exclude the time of extracting source files.

Fig. 14 presents the results of the experiment. It is evident that varying the number of the maximum search step does not significantly impact time efficiency. While there is a slight increase in time cost as the number of maximum search steps grows, the increase is not substantial. Similarly, varying the number of the maximum search steps does not significantly impact the F1-score. It increases from a maximum search step of 1 to 3, then decreases from 4 to 5. The reason for growth is pretty easy to understand. As the number of the maximum search step grows, SEM-SZZ can expand the search scope and analyze more basic blocks. However, as mentioned earlier, most bug-inducing commits can be identified by tracing back nearby lines. Therefore, we can assume that if a basic block is far from the added lines, it is unlikely to contain lines that can be traced back to find bug-inducing commits. Consequently, if the parameter N is set too large and the search scope is too broad, there is a high probability of including irrelevant statements in the identified buggy statements, even if these statements have data flow or control flow relationships with the added lines. Finally, the inclusion of irrelevant statements in the identified buggy statements can adversely affect the bug-inducing commit localization process, resulting in false results.

TABLE VI
THE PERFORMANCE COMPARISONS BETWEEN SEM-SZZ AND BUGNNINGS

| Approach | Precision | Recall | F1-score |
|---|---|---|---|
| BUGNNINGS | 0.51 | 0.35 | 0.41 |
| BUGNNINGS* | 0.51 | 0.50 | 0.50 |
| SEM-SZZ | **0.68** | 0.66 | **0.67** |

A typical example is commit 5574d329044 [41]. When the parameter N is set to 3, SEM-SZZ successfully identifies the correct bug-inducing commit. However, when the parameter N is increased to 5, SEM-SZZ fails to find the correct bug-inducing commit. This failure occurs because, with N set to 5, the search scope becomes too large and includes the statement `struct device *dev = &priv->udev->dev;` as one of the statements contributing to the bug. This statement, located over 100 lines away from the added line, is unrelated to the bug, despite its data flow relation with the added line.

## D. Comparison With Another Semantic-Based Approach

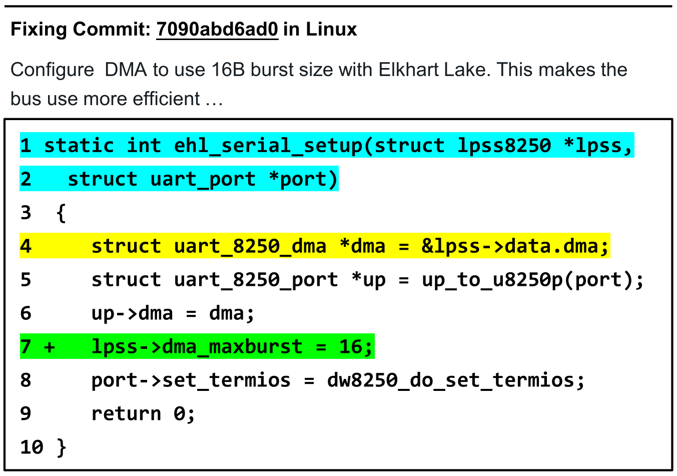
In this section, we compare SEM-SZZ with another semantic-based approach called BUGNNINGS [42]. This method first builds program dependence graphs for both the bug-fixing version and the version preceding the bug-fixing commit. It then computes the differences between these two versions of the program dependence graph. This helps identify the code statements that have a dependent relationship with the code changes in the bug-fixing commit. Finally, the method traverses the commit history to identify the earliest version that introduced all code statements with a dependence relationship to the changes in the bug-fixing commit.

The original implementation of BUGNNINGS is based on WALA, a Java program analysis infrastructure, and requires compiling the program. However, compiling and analyzing a large program, such as the Linux kernel, takes a very long time. Considering that our dataset contains tens of thousands of bug-fixing commits, strictly following the implementation in the paper would result in unacceptable time consumption.

To address this, we utilize a tool called Joern [43], which can build the program dependence graph (PDG) from the abstract syntax tree (AST). We first extract the file content of the two program versions and use Joern to build the PDG based on this content. Then, we compare the PDGs of the two versions to identify code statements that have a dependence relationship with the changes in the bug-fixing commit.

The experimental result is shown in Table VI. The original result can be seen in the second row of the table. However, during implementation, we found that Joern failed to generate PDGs for some test cases. Therefore, we filtered out these test cases and obtained the result labeled BUGNNINGS*. According to the table, SEM-SZZ outperforms BUGNNINGS in both precision and recall.

The fundamental reason for this is that comparing the differences in PDGs is not suitable in this scenario. For example, consider the commit 7090abd6ad0 [44] in Fig. 15. According to



**Fixing Commit: 7090abd6ad0 in Linux**

Configure DMA to use 16B burst size with Elkhart Lake. This makes the bus use more efficient …

```
1 static int ehl_serial_setup(struct lpss8250 *lpss,
2   struct uart_port *port)
3 {
4     struct uart_8250_dma *dma = &lpss->data.dma;
5     struct uart_8250_port *up = up_to_u8250p(port);
6     up->dma = dma;
7 +   lpss->dma_maxburst = 16;
8     port->set_termios = dw8250_do_set_termios;
9     return 0;
10 }
```

Fig. 15. A failing example of the BUGNNINGS approach.

the commit message, the bug occurs because the bug-inducing commit introduces the `dma` variable but does not set the burst size. In SEM-SZZ, the data flow of the `lpss` variable is compared, which correctly locates line 4, highlighted in yellow. Conversely, BUGNNINGS incorrectly locates the place where `lpss` is defined, which is line 1.

In fact, BUGNNINGS often tends to locate line numbers far from the added lines based on PDGs. As mentioned earlier, most bug-inducing commits can be found by tracing back the lines near the added lines. This is the one main reason for its poor performance. Another reason is its analysis of control flow dependence. For instance, a newly added `if` statement can have the control flow dependent relationship with many lines in the program, often including many irrelevant lines.

### E. Threats to Validity

**Internal Validity.** Threats to internal validity refer to causality, bias, and errors in the experiment. One potential threat is that we do not compile the entire source code of the program. Instead, we utilize the tree-sitter parser to implement fuzzy parsing [45], allowing us to analyze only part of the program. However, fuzzy parsing may not accurately reflect the behavior of the program, especially when the program contains many macros. This could potentially lead to errors when creating datasets and undermine the effectiveness of SEM-SZZ in comparison. Our implementation of BUGNNINGS faces the same issue, as we build program dependence graphs directly from abstract syntax trees. Another threat is the analysis of failure cases in the discussion. We only randomly selected a subset of failure cases for analysis which could also result in bias.

**External Validity.** Threats to external validity refer to the applicability of our approach to other scenarios. One potential threat is that we only implemented and evaluated our approach on the projects in the C programming language. This limitation arises due to the lack of ghost commits in other programming languages. This raises concerns about the portability of our approach to other programming languages. To enhance generalization, we evaluate our approach on a larger dataset than those used in previous studies. Another threat is that, in our dataset, each bug-fixing commit typically corresponds to only one bug-inducing commit, which may not reflect real-world scenarios. In the future, we plan to extend our approach to more programming languages and collect additional datasets to address these issues.

**Construct Validity.** Threats to construct validity refer to the accuracy of the measures used to represent the concepts in the study. Our approach might not perfectly encapsulate the complexities of real-world software development. For example, SEM-SZZ cannot handle test cases without changed lines in functions. Therefore, it is not applicable to the entire dataset. To mitigate this threat, we power a baseline with SEM-SZZ and apply it on the whole dataset when evaluating SEM-SZZ's effectiveness.

**Conclusion Validity.** Threats to conclusion validity refers to the accuracy of the conclusions drawn from the data. In our study, we draw conclusions based on a dataset from the Linux kernel, which may not fully represent other software development scenarios. Although we have analyzed the dataset and found that it includes both small and large commits, similar to those found in other development contexts, there may still be limitations in generalizing our findings. To address this, we plan to extend our approach to include other programming languages and projects in the future.

## VII. RELATED WORK

**SZZ algorithm Evaluation.** The SZZ algorithm and its variants have been the foundation in numerous software engineering studies, leading to extensive evaluations by researchers [33], [34]. Initially, evaluations often involved manual annotation, where researchers like Davies et al. [33] manually annotated 174 bugs across three repositories. However, manual annotation is time-consuming and may not always yield accurate results, as annotators may lack an in-depth understanding of the projects. Recognizing these challenges, many researchers have proposed developer-informed oracles as an alternative. For instance, Wen et al. [36] collected bug-inducing commits based on bug reports, while Rosa et al. [35] relied on commit messages for identification. Despite these advancements, one persistent issue has been the limited size of available datasets. Most existing datasets are relatively small, capturing only a fraction of the bug-inducing commits present in real-world projects. To address this limitation, Lyu et al. [20] collect a large dataset based on commit messages from the Linux kernel. This dataset, being considerably larger in scale, provides a more comprehensive view of bug-inducing commits in a real-world context.

**SZZ algorithm Application.** The original SZZ algorithm and its variants are widely used by researchers in various empirical studies within software engineering. These algorithms have been applied to investigate developer collaboration [46], code reviews [47], [48], technical debt [49], software vulnerability [17] and software quality [50]. For instance, Palomba et al. [51] employed the SZZ algorithm to explore the correlation between smelly code and code that introduces faults. Similarly, Bavota et al. [47] delved into the impact of the code review process on the likelihood of introducing a bug. Furthermore, the SZZ algorithm has also been used to determine the correct versions affected by a vulnerability in the National Vulnerability Database [17]. Another broad application of the SZZ algorithm is in just-in-time defect detection [52], where researchers aim to determine whether a commit introduces a defect. Datasets [37], [53], [54] used for just-in-time defect detection are often collected using the SZZ algorithm. Based on these datasets, many researchers have proposed models and evaluated their effectiveness. For instance, Kamei et al. [52] utilized the B-SZZ algorithm to identify bug-inducing commits and collected a large dataset for their study. Similarly, Fan et al. [37] investigated the impacts of different variants of the SZZ algorithm on defect detection.

**Factors leading to buggy commits.** Researchers have extensively explored the characteristics of bug-inducing commits, with a particular emphasis on the factors contributing to their production [2], [4], [6], [55], [56]. For instance, Śliwerski et al. [6] analyzed the time factor and discovered that commits made on Fridays are more prone to introducing bugs. Other studies have delved into the developer factor, including developers' coding habits and experience. Eyolfson et al. [55] found that developers who commit code daily are less likely to introduce bugs, while Bernardi et al. [56] discovered that developers who communicate frequently with their colleagues tend to produce fewer bugs. Additionally, Rahman et al. [4] identified that developers with experience in the target file are less likely to introduce buggy commits.

## VIII. Conclusion and Future Work

In this study, we observe that the majority of bug-inducing commits for bug-fixing commits with only added lines can be effectively located by tracing back the nearby lines of the unmodified lines. To further enhance precision, we introduce our approach, SEM-SZZ. Our approach leverages data flow analysis and control flow analysis to compare the states between the two versions of the program, identifying their difference and generating the statements contributing to the bug. Subsequently, we trace back the buggy statements to identify candidate commits. Finally, we employ line similarity comparison to pinpoint the commit that initially introduced all the buggy statements, marking it as the bug-inducing commit. The result shows that our proposed method significantly improves the precision and F1-score of the baselines in all scenarios, regardless of whether the bug-fixing commit contains deleted lines. In the future, we plan to extend our work to other programming languages and collect more high-quality datasets. Additionally, current methods overlook commit messages when locating bug-inducing commits. We intend to leverage large language models to interpret commit messages and code changes, using this information to filter out irrelevant lines detected by baselines such as A-SZZ. Furthermore, our current approach relies on line similarity to identify buggy commits. We aim to replace this with large language models. We hope this can maintain the high recall of A-SZZ while gaining SEM-SZZ's high precision.

## Data Availability

The replication package, which includes the source code, datasets, and prediction results, can be found at https://figshare.com/s/a25207be10150e2f0177.

## References

[1] 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.

[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 *Proc. IEEE 12th Int. Work. Conf. Source Code Anal. Manipulation*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 104–113.

[3] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs?" in *Proc. Int. Workshop Mining Softw. Repositories*, 2006, pp. 173–174.

[4] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 491–500.

[5] 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.,* Piscataway, NJ, USA: IEEE Press, 2011, pp. 191–200.

[6] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM Sigsoft Softw. Eng. Motes*, vol. 30, no. 4, pp. 1–5, 2005.

[7] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, "Automatic identification of bug-introducing changes," in *Proc. 21st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE'),* Piscataway, NJ, USA: IEEE Press, 2006, pp. 81–90.

[8] 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.

[9] 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. (SANER),* Piscataway, NJ, USA: IEEE Press, 2018, pp. 380–390.

[10] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2, Piscataway, NJ, USA: IEEE Press, 2015, pp. 99–108.

[11] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, Dec. 2018.

[12] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2012, pp. 200–210.

[13] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? The ManySStuBs4J dataset," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 573–577.

[14] 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. Symp. Found. Softw. Eng.*, 2021, pp. 1459–1462.

[15] M. Tufano, G. Bavota, D. Poshyvanyk, M. Di Penta, R. Oliveto, and A. De Lucia, "An empirical study on developer-related factors characterizing fix-inducing commits," *J. Softw.: Evol. Process*, vol. 29, no. 1, 2017, Art. no. e1797.

[16] B. Chen and Z. M. Jiang, "Extracting and studying the logging-code-issue-introducing changes in Java-based large-scale open source software systems," *Empirical Softw. Eng.*, vol. 24, pp. 2285–2322, Mar. 2019.

[17] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 2352–2364.

[18] L. Tang, L. Bao, X. Xia, and Z. Huang, "Neural szz algorithm," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2023, pp. 1024–1035.

[19] 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.

[20] Y. Lyu, H. J. Kang, R. Widyasari, J. Lawall, and D. Lo, "Evaluating SZZ implementations: An empirical study on the linux kernel," 2023, *arXiv:2308.05060*.

[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] "The commit of the motivation example," [Online]. Available: Accessed: Mar. 15, 2024. https://github.com/torvalds/linux/commit/e9838bd5116

[23] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: An empirical analysis," *Empirical Softw. Eng.*, vol. 26, pp. 1–45, Apr. 2021.

[24] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, 1984.

[25] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1456–1468.

[26] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[27] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—A formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[28] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[29] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: A model to identify how bugs are introduced in software components," *Empirical Softw. Eng.*, vol. 25, pp. 1294–1340, Feb. 2020.

[30] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1091–1095, Jun. 2007.

[31] "tree-sitter," 2024. [Online]. Available: https://github.com/tree-sitter/tree-sitter

[32] V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri, "Openszz: A free, open-source, web-accessible implementation of the SZZ algorithm," in *Proc. 28th Int. Conf. Program Comprehension*, 2020, pp. 446–450.

[33] 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. 1, pp. 107–139, 2014.

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

[35] G. Rosa et al., "Evaluating SZZ implementations through a developer-informed oracle," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 436–447.

[36] 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.*, 2019, pp. 326–337.

[37] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by SZZ on just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 47, no. 8, pp. 1559–1586, Aug. 2021.

[38] Line similarity failure example. Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/torvalds/linux/commit/15273ffd7ef

[39] A failing example. Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/torvalds/linux/commit/072b30642f9

[40] An example to show the effectiveness of program slicing. Accessed: Aug. 13, 2024. [Online]. Available: https://github.com/torvalds/linux/commit/bfab7c8ff89

[41] An example to show the impact of the maximum search step. Accessed: Mar. 15, 2024. [Online]. Available: https://github.com/torvalds/linux/commit/5574d329044

[42] V. S. Sinha, S. Sinha, and S. Rao, "Buginnings: Identifying the origins of a bug," in *Proc. 3rd India Softw. Eng. Conf.*, 2010, pp. 3–12.

[43] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy,* Piscataway, NJ, USA: IEEE Press, 2014, pp. 590–604.

[44] A failing example of the BUGNNINGS approach. Accessed: Aug. 13, 2024. [Online]. Available: https://github.com/torvalds/linux/commit/7090abd6ad0

[45] R. Koppler, "A systematic approach to fuzzy parsing," *Softw.: Pract. Experience*, vol. 27, no. 6, pp. 637–649, 1997.

[46] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante, "The relation between developers' communication and fix-inducing changes: An empirical study," *J. Syst. Softw.*, vol. 140, pp. 111–125, Mar. 2018.

[47] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME),* Piscataway, NJ, USA: IEEE Press, 2015, pp. 81–90.

[48] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME),* Piscataway, NJ, USA: IEEE Press, 2015, pp. 111–120.

[49] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Proc. IEEE 23Rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 1, Piscataway, NJ, USA: IEEE Press, 2016, pp. 179–188.

[50] B. Çaglayan and A. B. Bener, "Effect of developer collaboration activity on software quality in two large scale projects," *J. Syst. Softw.*, vol. 118, pp. 288–296, Mar. 2016.

[51] P. Fabio et al., "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," in *Proc. 40th Int. Conf. Softw. Eng., (ICSE),* New York, NY, USA: ACM, 2018, pp. 482–482.

[52] 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.

[53] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2013, pp. 279–289.

[54] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 560–560.

[55] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 153–162.

[56] 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.,* Piscataway, NJ, USA: IEEE Press: IEEE, 2012, pp. 139–148.

**Lingxiao Tang** is currently working toward the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University. He is currently under the supervision of Prof. Lingfeng Bao. His research mainly focuses on software defects and vulnerabilities.

**Chao Ni** received the Ph.D. degree in computer software and theory from Nanjing University, China, in 2020. Currently, he is an Associate Professor with the School of Software Technology, Zhejiang University. His research focuses on software quality assurance, including software testing, and vulnerability analysis.

**Qiao Huang** received the Ph.D. degree from the College of Computer Science and Technology, Zhejiang University, in 2020. Currently, he is an Associate Professor with the School of Computer Science and Technology, Zhejiang Gongshang University, China. His research interests include mining software repositories and empirical software engineering.

**Lingfeng Bao** received the B.E. and Ph.D. degrees from the College of Software Engineering, Zhejiang University, in 2010 and 2016. Currently, he is an Associate Professor with the State Key Laboratory of Blockchain and Data Security, Zhejiang University. His research interests include mining software repositories, AI4SE, and software security.