SemBIC: Semantic-Aware Identification of Bug-Inducing Commits

XIAO CHEN, Hong Kong University of Science and Technology, China
HENGCHENG ZHU, Hong Kong University of Science and Technology, China
JIALUN CAO*, Hong Kong University of Science and Technology, China
MING WEN, Huazhong University of Science and Technology, China
SHING-CHI CHEUNG*, Hong Kong University of Science and Technology, China

Debugging can be much facilitated if one can identify the evolution commit that introduced the bug leading to a detected failure (aka. bug-inducing commit, BIC). Although one may, in theory, locate BICs by executing the detected failing test on various historical commit versions, it is impractical when the test cannot be executed on some of those versions. On the other hand, existing static techniques often assume the availability of additional information such as patches and bug reports, or the applicability of predefined heuristics like commit chronology. However, these approaches are ineffective when such assumptions do not hold, which are often the case in practice. To address these limitations, we propose SemBIC to identify the BIC of a bug by statically tracking the semantic changes in the execution path prescribed by the failing test across successive historical commit versions. **Our insight is that the greater the semantic changes a commit introduces concerning the failing execution path of a target bug, the more likely it is to be the BIC.** To distill semantic changes relevant to the failure, we focus on three fine-grained semantic properties. We evaluate the performance of SemBIC on a benchmark containing 199 real-world bugs from 12 open-source projects. We found that SemBIC can identify BICs with high accuracy – it ranks the BIC as top 1 for 88 out of 199 bugs, and achieves an MRR of 0.520, outperforming the state-of-the-art technique by 29.4% and 13.6%, respectively.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Bug-inducing Commits, Semantic Analysis, Software Testing

ACM Reference Format:
Xiao Chen, Hengcheng Zhu, Jialun Cao, Ming Wen, and Shing-Chi Cheung. 2025. SemBIC: Semantic-Aware Identification of Bug-Inducing Commits. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE062 (July 2025), 23 pages. https://doi.org/10.1145/3715781

1 Introduction

During software evolution, developers may inevitably introduce bugs into software systems. Identifying the *commits* that introduce bugs (***bug-inducing commits***, or ***BICs*** in short) facilitates debugging and repair [28, 45–47], thus attracting interest from both academia and industry.

A dynamic approach to running test suites against all commits is, in theory, a solution to identifying BICs. However, the nontrivial overhead of running tests and the incompatible or

*Shing-Chi Cheung and Jialun Cao are the corresponding authors.

Authors' Contact Information: Xiao Chen, Hong Kong University of Science and Technology, Hong kong, China, xchenfu@cse.ust.hk; Hengcheng Zhu, Hong Kong University of Science and Technology, Hong Kong, China, hzhuaq@connect.ust.hk; Jialun Cao, Hong Kong University of Science and Technology, Hong Kong, China, jcaoap@cse.ust.hk; Ming Wen, Huazhong University of Science and Technology, Wuhan, China, mwenaa@hust.edu.cn; Shing-Chi Cheung, Hong Kong University of Science and Technology, Hong Kong, China, scc@cse.ust.hk.

outdated build/execution dependencies prevents adopting this solution in practice. In addition, compiling and executing test code on historical versions are reported to be more challenging (the success rate is one-fourth compared with compiling source code [26]. While the Bisection algorithm and its variance [2, 9] adopt a binary search strategy to prune the search space for BICs, they do not address the issues like dependency incompatibility or dependency unavailability. To get around the difficulties encountered by dynamic approaches, static techniques are proposed to identify BICs based on extra information such as *patches* [4, 11, 12, 23, 39] (i.e., the fix for the given test failure), *bug reports* [31], and predefined heuristics based on commit *timestamps* [2]. However, such information may not always be available or applicable, hindering the adoption of these static techniques. In addition, using the information of commit time could be misleading and point to an innocent commit (see Section 2 for more details). These approaches are ineffective when the prerequisites are not met, which is often the case in practice.

Among these existing techniques, Bisection (i.e., dynamic test execution combined with binary search) is theoretically ideal but often impractical for automation. Other static techniques attempt to address the limitation of dynamic approaches, but they do not look into the root cause of the test failure. Therefore, we seek a practical workaround for the BIC identification task that addresses the issue (i.e., identifying BIC of the given bug) from the root cause of the bug. Essentially, the shift from a passing to a failing test stems from semantic changes causing behavioral differences along the execution path of the failing test (i.e., the failing path). Thus, we propose to use static semantic analysis to approximate dynamic execution. The insight is that a BIC typically introduces significant semantic changes concerning the failing path, thus leading to test failure. In other words, ***the more a commit contributes to the failing path, the more likely it is a BIC***.

To achieve this goal, the core challenge lies in distilling code semantic changes concerning the failing path from each commit. First, *distilling the semantic changes relating to the failing path for effective BIC identification is yet to be investigated*. Though there is existing work utilizing semantic analysis for different tasks (e.g., change impact analysis [33], code clone detection [37, 48]), these methods are not designed for capturing the semantic differences along test execution paths that result in divergent test outcomes in order to identify BICs. Second, *deciding what kinds of semantic properties should be tracked for precise BIC identification remains unknown.* Not all code changes made by commits should be blamed. For example, a variable renaming or code structure reorganization is unlikely to introduce a bug, while a change in branch condition (e.g., `if (a > b)` is changed to `if (a == b)`) is more likely to introduce a bug. Noise from code changes that are less likely to be at fault may reduce the accuracy of BIC identification.

To address the above challenges, inspired by prior studies [20, 24, 53] utilizing program semantics, we propose to track the code changes with three semantic properties: (i) ***path constraints*** (e.g., a change from `if (a > b)` to `if (a == b)`), (ii) ***method invocations*** (e.g., a change from `a.getMax()` to `a.getMin()`), and (iii) ***return statement*** (e.g., a change from `return NaN` to `return 0`)). On top of these three properties, we then propose our approach, SemBIC, a code-change-aware BIC identification framework that does not rely on dynamic execution and extra information like bug reports or patches. In particular, given a failing test, SemBIC reduces the search space by filtering out the irrelevant commits according to the failing test, then ranks candidate commits according to how much they change the semantics related to the test failure. Our goal is to show the feasibility of our insight, and one could easily extend SemBIC to embrace more semantics representations.

We evaluated SemBIC with a benchmark consisting of 199 real-world bugs from 12 open-source projects. Overall, the evaluation results show that our technique can identify BICs with high accuracy. Specifically, SemBIC ranks the real BIC as top-1 for 88 bugs out of the total 199 and achieve an MRR of 0.520, outperforming the state-of-the-art (SOTA) BIC identification techniques by 29.4% and 13.6%. Additionally, SemBIC stands to gain from more refined and precise fault

```
1 @Test
2 public void testAddNaN() {
3     Complex x = new Complex(3.0, 4.0);
4     Complex z = new Complex(1, nan);
5     Complex w = x.add(z);
6     Assert.assertTrue(Double.isNaN(w.getReal()));
7     Assert.assertTrue(Double.isNaN(w.getImaginary()));
8 }
```

Listing 1. Failing Test of Subject Math-53

```
1 public Complex add(Complex rhs)
2     throws NullArgumentException {
3     MathUtils.checkNotNull(rhs);
4 +   if (isNaN || rhs.isNaN) {
5 +       return NaN;
6 +   }
7     return createComplex(real + rhs.getReal(),
8         imaginary + rhs.getImaginary());
9 }
```

Listing 2. Patch of Subject Math-53

localization (FL) techniques in the future, as our experiments indicate its performance enhances with the improvement of FL results. Our evaluation results also confirm the contribution of all the three semantic properties and the importance of a multi-dimensional semantic analysis in boosting BIC identification accuracy.

In summary, we made the following key contributions in this paper:

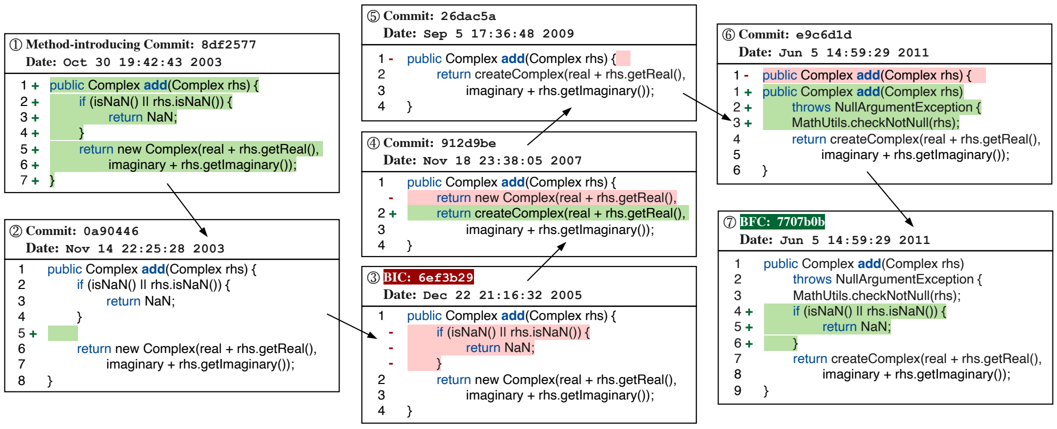
- **Originality**: We introduce SemBIC, a technique that identifies BICs by statically tracking the code semantic changes along the execution path prescribed by failing tests across successive historical commit versions.
- **Approach**: To distill code changes relevant to the failure, we propose to focus on three fine-grained semantic properties, so as to characterize program behavior changes along the path prescribed by the failing tests that lead to different test outcomes.
- **Evaluation**: We conduct a series of experiments to evaluate the effectiveness of SemBIC across subjects from various projects, comparing it with several baseline techniques including the SOTA. The evaluation results show that our technique can achieve overall high accuracy in identifying BICs. SemBIC ranks the real BIC to top-1 for 88 bugs out of the total 199 and achieve an MRR of 0.520, outperforming the SOTA BIC identification technique by 13.6%.

2 Motivating Example

In this section, we illustrate our insights with a real-world example in Defects4J [22]. Listing 1 shows the failing test that reveals a real bug in `Math-53`[1] from project `commons-math`[2]. The test case `testAddNaN` is checking if `Complex.add` correctly handles the case where any of the operands is `NaN`. Specifically, it checks if the method `add` returns `NaN` when it receives an `NaN` as an argument. The fault is caught at Line 6 by an assertion. To fix this bug, developers added an early return in Lines 4-6 of method `add` to handle the `NaN` case as shown in Listing 2.

Figure 1 shows the evolution of the buggy method `add`. The method was first introduced by commit ① `8df2577` with a correct implementation. The bug was later introduced by the BIC③

[1]https://program-repair.org/defects4j-dissection/#!/bug/Math/53
[2]https://github.com/apache/commons-math



```
① Method-introducing Commit: 8df2577
   Date: Oct 30 19:42:43 2003
1 + public Complex add(Complex rhs) {
2 +     if (isNaN() || rhs.isNaN()) {
3 +         return NaN;
4 +     }
5 +     return new Complex(real + rhs.getReal(),
6 +         imaginary + rhs.getImaginary());
7 + }
```

```
② Commit: 0a90446
   Date: Nov 14 22:25:28 2003
1     public Complex add(Complex rhs) {
2         if (isNaN() || rhs.isNaN()) {
3             return NaN;
4         }
5 +
6         return new Complex(real + rhs.getReal(),
7             imaginary + rhs.getImaginary());
8     }
```

```
③ BIC: 6ef3b29
   Date: Dec 22 21:16:32 2005
1     public Complex add(Complex rhs) {
  -       if (isNaN() || rhs.isNaN()) {
  -           return NaN;
  -       }
2         return new Complex(real + rhs.getReal(),
3             imaginary + rhs.getImaginary());
4     }
```

```
④ Commit: 912d9be
   Date: Nov 18 23:38:05 2007
1     public Complex add(Complex rhs) {
  -       return new Complex(real + rhs.getReal(),
2 +       return createComplex(real + rhs.getReal(),
3             imaginary + rhs.getImaginary());
4     }
```

```
⑤ Commit: 26dac5a
   Date: Sep 5 17:36:48 2009
1 -   public Complex add(Complex rhs) {
2         return createComplex(real + rhs.getReal(),
3             imaginary + rhs.getImaginary());
4     }
```

```
⑥ Commit: e9c6d1d
   Date: Jun 5 14:59:29 2011
1 -   public Complex add(Complex rhs) {
1 +   public Complex add(Complex rhs)
2 +       throws NullArgumentException {
3 +       MathUtils.checkNotNull(rhs);
4         return createComplex(real + rhs.getReal(),
5             imaginary + rhs.getImaginary());
6     }
```

```
⑦ BFC: 7707b0b
   Date: Jun 5 14:59:29 2011
1     public Complex add(Complex rhs)
2         throws NullArgumentException {
3         MathUtils.checkNotNull(rhs);
4 +       if (isNaN() || rhs.isNaN()) {
5 +           return NaN;
6 +       }
7         return createComplex(real + rhs.getReal(),
8             imaginary + rhs.getImaginary());
9     }
```

Fig. 1. Evolution of the Buggy Method in Subject Math-53.

6ef3b29, which incorrectly removes the branch that handles NaN, in December 2005. After that, the buggy method was subsequently modified by various commits before it was fixed by bug-fixing commit (BFC) 7707b0b in June 2011, which is over five years after the introduction of BIC.

During software evolution, two main factors render the Bisection algorithm inapplicable to this bug. First, test dependencies (e.g., the private variable nan used in line 4 in Listing 1) related to the assertion that actually triggers this bug (i.e., line 6 in Listing 1) are not available before commit version 6ef3b29. Second, the upgrade of unit tests to JUnit 4 [21] in March 2011 with commit a4bbdaf (i.e., a commit submitted between commit ⑤ and commit ⑥) complicates the automation of the Bisection algorithm. On the other hand, since the SOTA technique FONTE [2] leverages a temporal heuristic to identity BIC, it mistakenly identifies commit e9c6d1d as Top-1 suspicious BIC because it is the latest commit that modified the method add.

To address such limitations, we employ code analysis statically instead of applying direct dynamic test execution against historical commits. Furthermore, our static analysis focuses on semantic changes related to the test failure for BIC identification, rather than relying on predefined rules based on ancillary information. Intuitively, SEMBIC estimates if the outcomes (i.e., pass/fail) of the failing test between two consecutive program versions would differ based on their semantic differences along the failing test's execution path. If the differences are significant, the test outcomes of the failing test on these two versions are estimated to be different. By estimating the committed versions in a reverse temporal order, SEMBIC identifies the BIC for the failing test.

Fundamentally, the transition from a passing (or fail-to-compile) to a failing test is caused by the changes in semantic information along the test execution path. Although dynamic test execution can precisely capture semantic information, as previously explained, it is impractical to implement in real-world scenarios. Thus, we aim to develop a static mechanism to estimate the semantic differences without program execution. Inspired by previous research works [20, 24, 53] that leverage program semantics tailored for different tasks and analysis systems, we focus on semantic properties that are more likely to change program dynamic behaviors along a specific test execution path. In the example shown in Figure 1, on the correct commit ② version, the assertion triggering the bug (Line 6 of test testAddNaN in Listing 1) passes because variable w is assigned to NaN with the correct implementation of method add. In the correct method add implemented in correct commit ② version, given the input Complex(1, nan) (according to Line 4-5 in Listing 1), the second ***constraint*** of the if statement (i.e., Line 2 in commit ②) is satisfied. The constraints

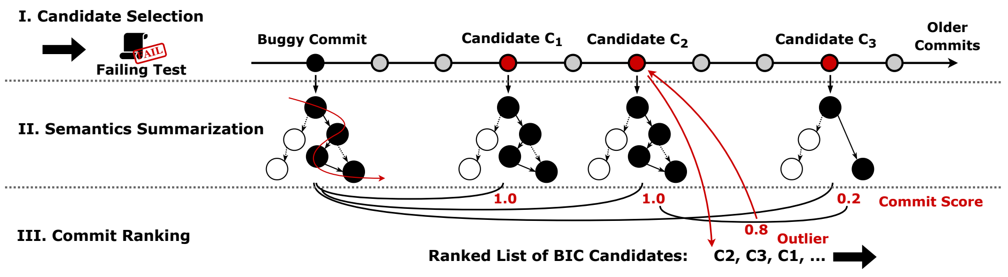


Fig. 2. Overview of SemBIC.

are implemented by the ***invocation*** `isNaN`. Then the program ***return***s `NaN` and exits method `add`. After committing BIC ③, the buggy method `add` wrongly ***return***s a new object (line 2-3 in BIC ③ in Figure 1) without ***constraint*** validation by the method ***invocation*** `isNaN` (Line 2 in commit ② in Figure 1), leading to the assertion failure. Based on these observations, SemBIC characterizes the semantics prescribed by the failing execution path based on three properties: the constraints imposed by the logical expressions evaluated, the methods invoked, and the return statement visited by the path.

In summary, we observe that failure-relevant semantic information can effectively characterize program behavior to approximate the test outcome without dynamic execution. We propose to estimate whether the outcomes of a failing test between two consecutive program versions will differ, based on significant semantic differences with respective to three properties (i.e., path constraints, method invocations and return statement) along the failing test's execution path. SemBIC identifies the BIC responsible for the failing test by performing this estimation on committed versions in reverse chronological order. The BIC is likely to be the commit that introduces the most significant semantic changes along the failing path.

3 Methodology

Given a set of commits and tests (including a few *failing tests* that trigger a specific bug and a set of *passing tests*), SemBIC outputs a prioritized list of suspicious commits. As illustrated in Figure 2, it comprises three phases:

- **Phase I: Candidate Selection.** SemBIC is designed to rank the commits based on their suspiciousness. Since not all the commits are relevant to the bug, including the irrelevant commits will not only increase the computation resource taken by SemBIC but also increase developers' burden in checking the resulting rank. Therefore, in the first phase, we select the candidate commits that are relevant to the failing test. This can help narrow down the search space of SemBIC and provide developers' with a cleaner rank.
- **Phase II: Semantics Summarization.** Due to the difficulty of running the failing test on historical versions [26] in practice, we opted for a static approach to estimate the test execution on historical versions. Specifically, for each of the relevant commits identified in Phase I, we generate a summary of its semantics, which captures how the behavior of the program responds to test inputs, offering a high-level overview of the program behavior rather than syntactic characteristics.
- **Phase III: Commit Ranking.** Based on the insight that the BIC contributes the most to the test failure, we track the semantic changes of each commit by calculating the differences in the semantic summary from its previous version. We quantify the semantics difference with the

edit distance between the tree representations of the semantic summaries. Finally, we rank the suspiciousness of each commits based on their semantic difference and fault localization scores.

3.1 Candidate Selection

Given a set of commits and tests that can trigger a bug, it is unnecessary to analyze all the commits in detail. Thus, we first select commits that related to the failing tests to narrow down the search space. We achieve this in two steps, i.e., we first identify the suspicious methods relating to the failing test, then identify the suspicious commits that relate to these suspicious methods from all commits.

Suspicious Method Identification. We first run the tests on a commit where the tests fail in order to gather the execution path, (i.e., failing path). Along the path, we can obtain all the methods invoked by the failing tests (i.e., suspicious methods). However, the number of methods invoked by failing tests could be huge, and not all of them should be blamed. For example, if a method happens to be invoked by a falling test while all the passing tests extensively invoke it, then it is unlikely to be responsible for the bug. So we utilize a classic FL approach (i.e., spectrum-based fault localization, SBFL) [1, 13, 30, 44, 50, 52] to further identify more suspicious methods.

To achieve more precise localization to the specific bug, it is important to select test cases (including both passing and failing tests) carefully [5]. Note that there could be multiple bugs in commit versions (i.e., programs); each time, we only focus on the bug triggered by the given failing tests. Since SBFL compares the proportion of methods' participation in failing/passing tests, the suspiciousness of buggy methods could be underestimated by involving too many irrelevant passing tests. Thus, we select the passing tests that are more likely to test the methods similar to those invoked by the failing tests. In particular, according to the set of given failing tests, we first collect all the classes loaded by these failing tests (i.e., suspicious classes). By doing so, we can exclude the classes irrelevant to the failure. Then, we consider two criteria for relevant passing test selection. First, according to the naming convention, for each loaded class, we keep the passing test classes whose name is a permutation of the loaded class name and `Test`. For example, suppose the loaded class is named `ClassA`, then either `TestClassA` or `ClassATest` will be kept if they are passing tests. Note that the selected tests should be in the same package as the suspicious classes. Second, according to the suspicious classes, it is possible that some tests that do not obey the naming convention. To cover this situation, we also keep the passing tests that load any of the suspicious classes. Note that this step scans the import classes and checks whether they exist statically, for efficiency.

After selecting relevant passing tests, the suspicious scores of the suspicious methods can be calculated against the given failing tests and the selected passing tests. In particular, for each method, we profile each statement by a suspicious score and use the highest suspicious score within that method to represent the suspiciousness of the method (i.e., the suspicious score of the method). We use Ochiai formula [1, 30, 44, 50, 52] to calculate the suspicious score. Since FL is not the focus of this work, one can easily turn to more advanced formulas or FL techniques to improve SemBIC. Finally, we keep the methods whose suspicious scores are higher than zero as the suspicious methods.

Suspicious Commit Identification. After suspicious methods are identified, the commits that involve the changes to these methods are naturally to be regarded as suspicious commits. To achieve this, we conduct a two-step selection to effectively filter out irrelevant commits.

First, in order to enhance the overall efficiency, we select commits that modified any of the Java files containing definitions of Java classes loaded by failing tests. This step efficiently reduces the search space by quickly filtering commits based on Java source file name matching, utilizing the loaded class information collected from the previous failing path collection. Second, we eliminate

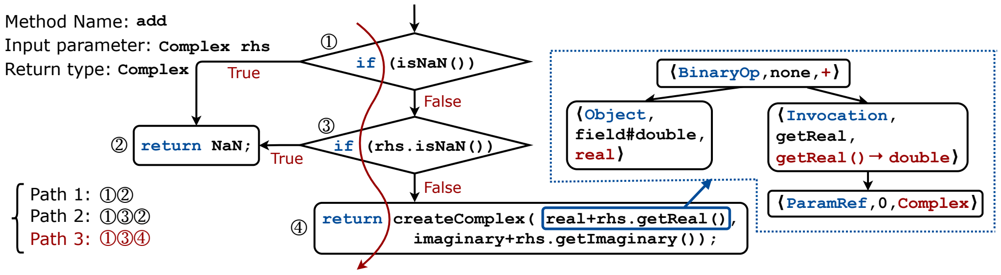

Method Name: add
Input parameter: Complex rhs
Return type: Complex
① if (isNaN())
True
False
② return NaN;
③ if (rhs.isNaN())
True
False
④ return createComplex(real+rhs.getReal(), imaginary+rhs.getImaginary());
Path 1: ①②
Path 2: ①③②
Path 3: ①③④
⟨BinaryOp,none,+⟩
⟨Object, field#double, real⟩
⟨Invocation, getReal, getReal() → double⟩
⟨ParamRef,0,Complex⟩

Fig. 3. CFG of the Buggy Method in Subject Math-53 and an Example for Expression Representation.

commits that only modified uninvoked methods that are defined in Java classes loaded by failing tests or invoked methods with zero suspiciousness. From the gathered file-changing commits, we extract the modified methods in the changed Java source files to further select commits that modified the suspicious methods (i.e., methods along the failing path with non-zero suspiciousness).

3.2 Semantics Summarization

After selecting the candidate commits, SemBIC summarizes the semantics associated with the failing path for each commit version. We leverage semantic summaries to effectively capture the semantic differences that lead to different test execution outcomes between the failing path in the buggy version and the failing path in the candidate commit versions. They will be forwarded to the following phase for BIC identification by commit ranking with semantic changes.

Path semantics summarization. We maintain that path semantics are typically encapsulated by the control and data flow within the failing paths. The primary challenge lies in adopting a semantic-sensitive representation that ensures semantically-similar paths are closely aligned while those with dissimilar semantics remain distinctly separate. We consider three fine-grained control and data flow properties, including path constraints, method invocations and return statements, for path semantics summary generation.

- **Path Constraints.** The failing path constraints of a method are the branch conditions that are satisfied by the failing path in the method. They provide concrete information on the conditions under which the program will take the failing path. This is crucial for understanding a program's behavior and deducing its response to the input of the the failing test. Consequently, the collected path constraints help discern the semantic differences between the code associated with the failure from the buggy and commit versions. Specifically, SemBIC stores the collected path constraints in a list, and each constraint is represented by an abstract syntax tree of the constraint's logic expression. Figure 3 shows the control flow graph (CFG) of the buggy method in Subject Math-53. In Figure 4, we depict the path constraints along path 3 in the CFG of Figure 3. In this path, two constraints from basic blocks 1 and 3 are represented as trees, respectively. Since the program only runs along this path when both constraints are satisfied, we arrange the two constraint trees in a list in order as part of the path semantics summary. In the example in Figure 1, there is no constraint in the given buggy commit version ⑥ and BIC candidates ③④⑤, while there are path constraints in BIC candidate ②, which is actually a correct commit version before submitting BIC ③. Based on the differences in path constraints, SemBIC identifies BIC ③ as highly suspicious for introducing the bug.
- **Method Invocations.** Method invocations explicitly capture the hierarchy and dependencies between methods, including both control flow information of how execution jumps from one method to another and data flow information of how data is likely to flow from callers

to callees. Consequently, method invocation changes along the path are likely responsible for alterations in program behavior and are a crucial component of path semantics. Method invocations are also maintained by a list that records all method calls along the path. For each method invocation, we record its signature. And we further resolve the information of the corresponding object if we meet an instance invocation. The method invocation sequence for path 3 collected by SemBIC, is detailed in Figure 4. In the example in Figure 1, the code lines removed by the BIC ③ involve method invocations along the failing path. Along the failing path (i.e., the path that returns the newly-created `Complex` object), correct commits ①② before BIC ③ use two instance invocations of method `isNaN` for condition checking while buggy commits ③④⑤ do not.

- **Return Statements.** Return statements encapsulate the final output of a method, thereby influencing the data and control flow within the caller methods. It is common that the execution path that follows a method call depends on the value returned by that method. This return value can influence conditional branches in the calling function, thus shaping the program's control flow. Since cross-method path enumeration will encounter the problem of path explosion, which is especially common in complex project (e.g., `commons-closure`) where a failing assertion goes through thousands of methods, we conduct semantic summarization at the method level using intra-procedural analysis. Thus, return statements can serve as a complement for inter-procedural analysis. The content of each return statement is treated as an expression and can also be effectively represented by a tree, as illustrated in the example in Figure 4.

We performed intra-procedural data-flow analysis by resolving the definition of each variable for collecting semantics. Finally, the three properties are integrated together to represent the semantics of the failing path.

Path extraction and analysis. Since the path comprises method invocations, we begin by constructing a call graph of the failing test. Then, SemBIC analyzes methods that (1) are deemed suspicious (i.e., a method whose suspiciousness computed by existing FL technique is higher than zero) and (2) were modified by the current candidate commit, and generates a path semantics summary for each suspicious method. The call graph is constructed during dynamic analysis, containing only executed method invocations, and is used in static analysis. During static analysis, SemBIC will not dive into a method if it is non-suspicious or unmodified by the analyzed commit. Hence, the path semantics summary of a commit version is structured by the path semantics summaries from each invoked method in the call graph of a failing test. The analysis process of these suspicious methods can be divided into two situations.

Dynamic path analysis for the methods in the given buggy program version. For the given buggy program, we have obtained the failing path via dynamic execution. So, we only need to analyze the determined failing path in each failure-related method. In the CFG of the buggy method in Subject `Math-53` shown in Figure 3, SemBIC focuses on the red-highlighted failing path, and collect the failing paths determined by execution for each failure-relevant method in the given buggy version. Note that we collected all unique paths traversed by test execution if the test invokes the same method multiple times.

Static path analysis for methods in the historical commit versions. For path-undetermined methods changed by candidate commits, we apply over-approximation by enumerating paths within failure-related methods. We use a depth-first search (DFS) algorithm to traverse all paths in the CFG of the failure-related method. Therefore, we construct a directed cyclic graph encompassing all possible paths. Each path starts from an entry block of the method and ends with an exit block (e.g., a `return` or `throw` statement). Taking the CFG in Figure 3 as an example, SemBIC enumerates 3

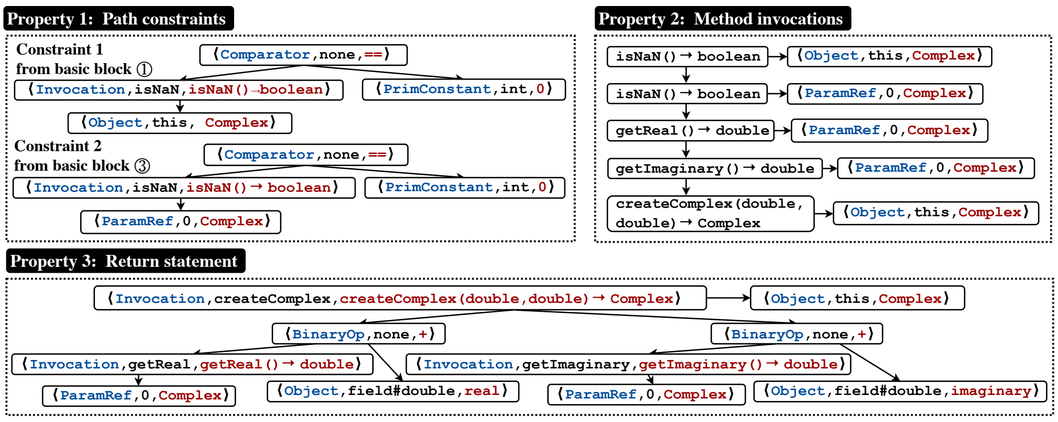

Property 1: Path constraints
Constraint 1 from basic block ①
⟨Comparator,none,==⟩
⟨Invocation,isNaN,isNaN()→boolean⟩
⟨PrimConstant,int,0⟩
⟨Object,this, Complex⟩
Constraint 2 from basic block ③
⟨Comparator,none,==⟩
⟨Invocation,isNaN,isNaN()→boolean⟩
⟨PrimConstant,int,0⟩
⟨ParamRef,0,Complex⟩
Property 2: Method invocations
isNaN()→boolean
⟨Object,this,Complex⟩
isNaN()→boolean
⟨ParamRef,0,Complex⟩
getReal()→double
⟨ParamRef,0,Complex⟩
getImaginary()→double
⟨ParamRef,0,Complex⟩
createComplex(double, double)→Complex
⟨Object,this,Complex⟩
Property 3: Return statement
⟨Invocation,createComplex,createComplex(double,double)→Complex⟩
⟨Object,this,Complex⟩
⟨BinaryOp,none,+⟩
⟨BinaryOp,none,+⟩
⟨Invocation,getReal,getReal()→double⟩
⟨Invocation,getImaginary,getImaginary()→double⟩
⟨ParamRef,0,Complex⟩
⟨Object,field#double,real⟩
⟨ParamRef,0,Complex⟩
⟨Object,field#double,imaginary⟩

Fig. 4. Semantic Properties for Path 3 in Fig. 3.

possible paths and gathers all as candidates. We limit each edge (i.e., transitions between basic blocks) to be traversed a maximum of once per path to handle loops efficiently. In summary, we collect a set of possible paths for each suspicious method changed by candidate commits in the historical versions.

Expression representation. After extracting failing paths for both the given buggy version and the historical commit versions, SemBIC preserves failure-relevant semantics by generating a summary for each path. To ensure soundness and maintain analytical feasibility, we opt for a relatively conservative way of over-approximating path semantics.

Expressions are fundamental constructs in programming languages, encompassing variables, predicates and more complex constructs. Inspired by previous work [20, 24, 53] that leverages program semantic information for different purposes, we introduce an informative Tree Structure for expression representation, specifically designed to capture fine-grained semantic features in BIC identification task, based on the following two insights. First, all expressions can ultimately be resolved into components consisting solely of constants, input parameters, objects, operators and invocations by tracing the def-use chains. Second, component attributes to record should be carefully customized according to different component types (e.g., we need to record the method name and parameter types of an invocation, while recording the symbol of a binary operator) for fine-grained semantics characterization. Accordingly, we categorized the components into 8 types as illustrated in Table 1. For each expression component, we represent it as a node described by three attributes: Type (T), Identifier (D) and Value (V), denoted by the following triple.

$$N := \langle T, D, V \rangle$$

Type belongs to the categorized eight component types. Identifier is typically the simple name of a method, the type of a primitive variable, or the simple class name of a class-based object. Values are defined separately for each node type to record certain path information. For instance, attribute Value of a primitive constant type (i.e., *PrimConstant* in Table 1) consists of the type of the constant variable (e.g., int, string, etc.) and the concrete value. Note that only nodes of the types *PrimConstant*, *Object*, *ParamRef*, *Invocation* and *NewExpr* can be leaf nodes.

The operands of non-leaf nodes can iteratively be decomposed into leaf nodes. We illustrate with a concrete example in Figure 3. For expression `real+rhs.getReal()` in the return statement shown at the bottom of the CFG, SemBIC first obtains its operator (i.e., +) with *BinaryOp* type as root node, and get the left operand (i.e., object `real`) as the left child node and the right operand

Table 1. Definition for Attributes of Expression Nodes.

Type	**Explanation**	**Identifier**	**Value**	**Example**
PrimConstant 🍃	Constants of primitive types.	Type of the constant.	Concrete value of the variable.	`double#1.2`, `string#"str_cur"`
ObjectRef 🍃	Reference of array, field and class objects.	Type of the object. For array and non-object field, we also record their base types.	The declaring class of the objects and non-object fields.	`array#float#org.apache.commons.math.complex. Complex`
ParamRef 🍃	Reference of method input parameters.	The index of the parameter.	Type of the parameter.	`0#string`
Comparator	Operators for value comparison.	-	Symbol of the comparator.	$>$, $\geq$, $<$, $\leq$, $==$, $!=$
BinaryOp	Operators for binary computation.	-	Symbol of the binary operator.	$+$, $-$, $*$, $/$, $\%$, $\ll$, $\gg$, $\&$, $\|$
UnaryOp	Operators for unary computation.	-	Symbol of the unary operator.	$+$, $-$, $++$, $--$, $!$, $\sim$
Invocation 🍃	Method invocations.	Simple Name of the invoked method.	Signature of the invoked method.	`org.apache.commons.math.complex.Complex : double getReal()`
NewExpr 🍃	Expressions to create new objects.	Type of the new object.	Base type of the object if available.	`array#int`

Nodes Type with 🍃 can be leaf nodes.
Though fields may not be objects, we include the references of fields in ObjectRef category.
For unary operators, + denotes unary plus, − denotes unary minus.

(i.e., expression `rhs.getReal()`) as the right child node. Then, SemBIC further resolves the right child node and generates a child leaf node of *ParamRef* type.

In this way, SemBIC generates a path semantic summary for each commit version by integrating the semantic information associated with the failing path from the three properties. For the given buggy version, the failing execution determines a single path. For candidate commit versions, we perform static analysis on all possible paths relevant to the failure and maintain a path list. The semantic summary for each commit, structured by the failing test call graph, comprises path semantic summaries from each method associated with the failing path. They are helpful in approximating the test execution on different commit versions. SemBIC then passes these semantic summaries to phase 3 for BIC identification via commit ranking.

3.3 Commit Ranking

With the path semantics summaries generated in the previous phase, SemBIC ranks candidate BICs according to their suspiciousness. The intuition is that the more significant the semantic changes along the failing path, the more likely the test outcomes between the buggy version and the candidate commit versions will differ. In other words, SemBIC prioritizes commits with more significant failure-relevant semantic changes as probable BICs. Specifically, SemBIC first calculates the semantic similarity between each candidate commit and the faulty commit in order to quantify the degree of semantic changes. Subsequently, it ranks the candidates according to the extent of semantic changes observed, giving priority to those with significant alterations.

Semantic Property Similarity Calculation. Since expressions that contribute to the semantics summary are represented in Tree Structures, we first explain how SemBIC computes the similarity between two trees. Given two trees T_1 and T_2, SemBIC calculates their similarity as follows.

$$\mathrm{simiT}(T_1, T_2) = (1 + \mathrm{TED}(T_1, T_2))^{-1} \tag{1}$$

$\mathrm{TED}(T_1, T_2)$ denotes Tree Edit Distance (TED) that quantifies the cost measured as the numerical value of effort required to transform tree T_1 into tree T_2. We follow Pawlik et al. [29] to implement TED calculation. Specifically, TED equals the least number of basic edit operations (i.e., node insertion, deletion, and modification) required to transform one tree into another. Each basic edit operation is associated with a numerical cost value, specifying the efforts the operation required in the tree transformation. In our design of SemBIC, the cost is 1 for insertion and deletion operations. For modification operations, if two nodes match precisely, no edit is required, and thus the cost is 0. If two nodes do not match, i.e., at least one of the node attributes does not match, each of the unmatched attribute pairs requires one-third of the operation to make them the same (i.e., the edit cost is thus $\frac{1}{3}$). The overall cost of transforming one tree into another is the sum of the costs of all operations required to make the two trees identical.

While the return statement of a path is denoted by only a Tree Structure, path constraints and method invocations are recorded by lists as there might be multiple along each path. Given two lists L_1 and L_2, SemBIC computes the list similarity $\mathrm{simiL}(L_1, L_2)$ by finding the optimized mapping (i.e., the mapping with the maximum sum of element pair similarity) between them. More specifically, we leveraged the Dynamic Programming algorithm designed for path constraint mapping and method invocation mapping proposed by Xie et al. [53] to compare the similarity between element pairs from two lists in order. Thus, the similarity between two lists is equal to the proportion of the optimized mapping length to the longer list length. Note that we use the longer list length as dominator for normalization, as the worst case is no element of the longer list can exactly match the shorter list.

As the three semantic properties we consider are represented in different formats (i.e., list and trees), SemBIC calculates the similarity of each property respectively. Constraints along the path are recorded using a list, with each constraint represented by a tree. Therefore, we compute the similarity between path constraints by combining the similarity measures of both data structures. Because method invocations are maintained by a list that records all invoked methods in the path, we compute the similarity between method invocation sequences by the list mapping algorithm [53]. For return statements represented with trees, the similarity is calculated by TED.

Commit Prioritization. SemBIC considers the average similarity across all three semantic properties as the similarity metric for a complete path summary. Given two paths p_1 and p_2, we denote the collected path constraint sets as PC_1 and PC_2, method invocation lists as M_1 and M_2 and return statement trees (i.e., the expression trees of the return value) as R_1 and R_2. The similarity between the two paths is calculated as follows.

$$\mathrm{simiP}(p_1, p_2) = \frac{1}{3}\left(\mathrm{simiL}(PC_1, PC_2) + \mathrm{simiL}(M_1, M_2) + \mathrm{simiT}(R_1, R_2)\right) \tag{2}$$

SemBIC utilize existing FL techniques to quantify the suspiciousness of methods, and further assign weights to the methods based on their suspiciousness. Given the i^{th} method m_i out of all n suspicious methods and p_j is the j^{th} path of all k possible paths in each suspicious method, we measure m_i's weight $\mathrm{weight}(m_i)$ regarding to its suspicious score $\mathrm{score}(m_i)$ and the method's rank $\mathrm{rank}(m_i)$ in descending order of suspicious scores. We explore four different weighting formulas, as outlined below.

$$\text{weight}(m_i) = \begin{cases} \text{score}(m_i) \\ 1/\text{rank}(m_i) \\ \text{score}(m_i)/\text{rank}(m_i) \\ \text{score}(m_i)^{\text{rank}(m_i)} \end{cases} \tag{3}$$

For the given buggy program version by commit c_f, the path p_f is determined by the failing test execution. For a program version by a candidate commit $c \subseteq C$, we can obtain a list of possible paths, i.e., $P = [p_1, p_2, ...]$. For each method in the failing test call graph, we approximate the path a candidate commit version would take given the failing test inputs by selecting the path among all k possible paths with the highest semantic similarity to the determined failing path. Taking the different suspiciousness of methods in the call graph into consideration, SemBIC scores each candidate by semantic similarity as follows.

$$\text{commitScore}(c) = \sum_{i=1}^{n} \left(\text{weight}(m_i) \cdot \max_{j=1}^{k} \text{simiP} \left(p_j, p_f \right) \right) \tag{4}$$

With commit scores, we further apply outlier analysis to rank the candidates. Intuitively, we consider commits that exhibit larger score disparities than their parent commits to reflect more significant semantic changes. In other words, instead of relying on absolute similarities, we rank commits based on the difference in similarity between them and their preceding commits to avoid going back too far to the early commits that probably have lower semantic similarities compared to the later commits. Based on this intuition, SemBIC generates a ranked list of candidate commits, ordered in descending suspiciousness of being the BIC.

Our path constraints are represented in product-of-sum form. The time complexity of our path comparison algorithm is $O(p^2t^2)$ where p is the number of maxterms in constraint and t is the expression tree size of each maxterm. This is because we leverage list mapping algorithm [53] with a complexity of $O(p^2)$ to compute the similarity between the lists of maxterms, and for each pair of maxterms, we utilize TED with a complexity of $O(t^2)$ to calculate the similarity on their expression trees.

4 Evaluation Design

This section presents the design of our evaluation of SemBIC.

4.1 Research Questions

To evaluate the effectiveness of SemBIC, we conducted a series of experiments to answer the following three research questions.

- **RQ1. How effective is SemBIC in BIC identification?** This research question evaluates the performance of SemBIC in BIC ranking. Towards this goal, we first investigate SemBIC's ranking results using different weighting formulas across various projects. Then, we compare SemBIC against three baseline ranking-based BIC identification techniques.
- **RQ2. How does the precision of FL affect the performance of SemBIC?** This research question assesses SemBIC's effectiveness in removing potential bottlenecks within FL techniques to explore the theoretical upper bound. To answer this question, we use Fonte, which relies on FL results, as the baseline and supply the precise FL information (i.e., the buggy method) to observe changes in ranking performance.
- **RQ3. How does each semantic property contribute to the BIC ranking?** This research question aims to further investigate the contribution of semantic analysis. To this end, we use

```
public final void translate(CharSequence input, Writer out) throws IOException {
    ...
    for (int pt = 0; pt < consumed; pt++) {
-       pos += Character.charCount(Character.codePointAt(input, pos));
+       pos += Character.charCount(Character.codePointAt(input, pt));
    ...
}
```

Listing 3. Patch of Subject Lang-6

```
public final void translate(CharSequence input, Writer out) throws IOException {
    ...
-   for(int j=0; j<consumed; j++) {
-       i += Character.charCount( Character.codePointAt(input, i) );
+   for (int pt = 0; pt < consumed; pt++) {
+       pos += Character.charCount(Character.codePointAt(input, pos));
    ...
}
```

Listing 4. Code Changes of BIC b4255e6 for Subject Lang-6 Labeled by Original Ground Truth

each semantic property separately and compare the BIC ranking accuracy to that achieved using the complete set of semantic properties.

4.2 Benchmark Construction

We collect evaluation subjects by combining two existing BIC datasets provided by An et al. [2] and Maes-Bermejo et al. [26] respectively. All subjects can be found in the Defects4J [22] dataset, which provides bug reports and corresponding patches. We exclude two old subjects whose commits, submitted before the BFCs, were managed using version 1.0 of the software project management tool Maven [17] due to the dependency and Java version incompatibility issue. This Maven version has a problem with the resolution of dependencies. As a result, we collected 199 subjects from 12 open source projects using Git as the version control system.

After further investigating the bug subjects and the corresponding BIC ground truth, we found there are two types of inaccuracy in the ground truth (i.e., the commit that is the actual BIC). First, there are cases when the BICs pinpointed by the original ground truth did not modify the buggy methods in the patches provided by Defects4J. Second, in cases where the code under test was wrongly implemented when first introduced until finally fixed, we observe inconsistencies in the criteria of ground truth BIC labeling. *For some subjects, the original ground truth identifies the commit that first introduced the buggy method with wrong implementation as the BIC.* For example, in subject `Math-93`, it labels commit `0a90446`, which first introduced the buggy method `factorial`, `factorialDouble` and `factorialLog` with wrong implementation, as BIC, even though several commits attempting to fix this bug were submitted before it was finally resolved. *For other subjects, the original ground truth designates the most recent commits that modified the buggy method prior to the BFC as the BICs.* For instance, in subject `Lang-6`, commit `b4255e6`, which is the latest to modify the buggy method `translate` before the BFC, is labeled as the BIC, despite several attempts to fix this bug in earlier commits. However, commit `b4255e6` should not be blamed for introducing the bug. This bug was fixed by rectifying the second parameter of method invocation `codePointAt` in the buggy method `translate` as shown in Listing 3. From the code changes shown in Listing 4, it is evident that commit `b4255e6` merely refactored two variables without changing the program's functionality, and therefore should not be blamed for introducing the bug. Thus, arbitrarily identifying the most recent commit that modified the buggy method as BICs is not a reliable way for establishing the BIC ground truth.

To mitigate such inconsistencies for unambiguous benchmark construction and fair evaluation, we manually validated the dataset. There are two potential threats in the validation process, i.e., biased data labeling criteria and mistakes in manual validation. To mitigate the first threat, we strictly followed the criteria proposed by Rodríguez-Pérez et al. named *perfect test* [35] to mentally perform the perfect test in historical versions, which was evaluated to be effective for constructing the BIC dataset. Specifically, given a *perfect test* passing on a BFC, we check all previous commits until finding the first commit where the test fails or the test cannot be compiled as the functionality under test has not been implemented yet. Therefore, we keep the original BIC ground truth `0a90446` for subject `Math-93` while changing the original BIC ground truth `b4255e6` to `a244767` (i.e., the commit that first introduced the buggy method `translate` with the wrong implementation) following the *perfect test* criteria. To mitigate the second threat (i.e., mistakes in manual validation), we discussed the relabeled instances during our meetings and reached consensus. We also released the dataset for validation.

4.3 Implementation Details

To evaluate our approach experimentally, we developed a prototype of SemBIC for Java projects. The implementation details are outlined below.

Failing test execution path collection. We collect the statement-level coverage using Cobertura [8] in order to record path information including loaded classes, invoked methods and executed basic blocks.

Commit tracking. We use Git [10] to track project commit history as all subjects in the benchmark are from projects using Git as version control system. For instance, we select the commits that modified any of the class files loaded by failing tests using Git command `git log <commit_ID> -- <class_file_name>`.

Code change extraction. From the gathered file-changing commits, we use JavaParser [19] to extract the changed methods in the changed class files to further select commits that modified the suspicious methods.

Path semantics summarization. We utilize Soot [41] to summarize path semantics at method-level granularity based on Jimple intermediate representation [40, 43]. Our analysis targets the Java bytecode to capture detailed semantic information precisely. Consequently, we initially compile the projects from Java source code into Java bytecode. During our experiments, we discovered that 94.4% of the commits could be automatically compiled using a software project management tool (i.e., Maven [17] or Ant [16]).

4.4 Experiment Setup

For evaluation, we use the following two metrics and four baselines. The details of each metric and baselineare as follows.

Evaluation metrics. We adopt the following two well-known evaluation metrics for evaluating the BIC ranking performance.

- Top-k Accuracy. The Top-k accuracy is equal to the proportion of subjects where the rankings of the BICs are within the top k positions. Achieving a high top-1 accuracy allows the results of a technique to be automatically utilized without the need for human intervention. Additionally, displaying top-10 results is considered beneficial in real-world debugging scenarios, since a prior FL study [3] indicates that showing top-10 results strikes an optimal balance between result quality and user interface (UI) succinctness. Thus, we choose $k = 1, 2, 3, 5, 10$ for this metric for a comprehensive evaluation.

- Mean Reciprocal Rank (MRR). This metric refers to the average of the reciprocal rank of the actual BIC. If the BIC appears at rank k in a list of returned results, the reciprocal rank is computed as $\frac{1}{k}$. Then, the MRR is calculated as $\frac{1}{n}\sum_{i=1}^{n}\frac{1}{k_i}$, where n is the number of all bug subjects and k_i is the BIC rank of the i^{th} bug subject. A higher MRR value indicates that the BIC typically appears closer to the top of the results list, thereby reducing the manual effort to conduct further investigations. In cases of tied rankings, the max-tiebreaker rule is applied to determine the rank. Effectively, if the scoring model is accurate, actual BICs should consistently receive higher scores and ranks compared to other commits.

Baselines. The bisection algorithm concludes by identifying a specific commit considered to be the BIC but does not rank commits. Therefore, we exclude it from our evaluation baselines. Thus, we use the following existing techniques as our evaluation baselines.

- Fonte. We first compare SemBIC with the SOTA BIC identification technique Fonte [2], proposed in 2023. This method scores and ranks candidate commits by integrating commit submission time with the suspiciousness of code elements.
- Bug2Commit. Bug2Commit [28] is currently the leading IR-based method for BIC identification. This technique ranks candidate commits via leveraging various features from commits and bug reports to pinpoint BICs. We utilize the implementation by An et al [15], which is based on Vector Space Model (VSM) [49] using the BM25 vectorizer [34] and the Ronin tokenizer [18]. Specifically, commit messages, modified files by patches, failing test stack traces with exception messages and the bug report content are utilized as the model input features. This information is provided by by Defects4J.
- FBL-BERT. This technique employs a context-aware architecture to address the BIC identification problem by evaluating the similarity between a query (i.e., the bug report) and a document (i.e., the software program) [7]. It leverages an adapted BERT-based model to localize the commits introducing changesets that caused the software bugs. Specifically, FBL-BERT evaluates the relevance of commits representing by the changesets they introduced to a given bug report by scoring them using the pre-trained model BERTOverflow [42]. For evaluation, we utilize the FBL-BERT model, fine-tuned with the provided training dataset using ARC changeset encoding [14] and Defects4J bug reports as input queries.
- SZZ Unleashed-RA-C. SZZ Unleashed-RA-C [27] is the SOTA implementation of an improved SZZ algorithm based on SZZ Unleashed [4], denoted as SZZ-RAC. The most important features that SZZ-RAC introduced include the ability to identify and handle refactoring changes, ignoring comments when identifying BICs, and the ability of using GitHub as the issue tracker. The input of this technique is a bug issue denoted with a bug ID, and the output is a set of BICs of the given bug. However, not all bug issues can be linked to a BFC. To better understand the advantages and disadvantages of SZZ algorithm, we exclude those bugs that are not linked to any BFC and calculate the performance of SZZ Unleashed-RA-C on the remaining bugs, denoted as SZZ-RA-C*. In addition, since SZZ algorithm generates a set of BICs without providing any prioritization, we follow the experiment settings of An et al. [2] to randomly rank the BICs in the set. So the expected rank of the BIC is $\frac{n+1}{2}$ when there are n commits in the output commit set.

5 Evaluation Results

In this section, we present the evaluation results with respect to each research question.

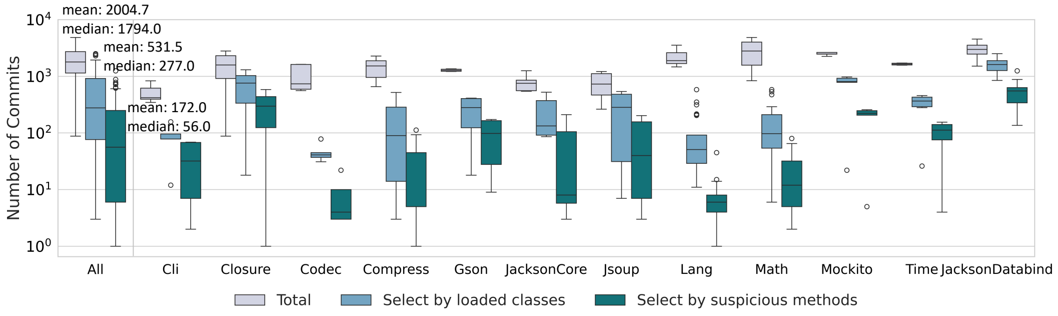


Fig. 5. Comparison between the Number of Commits after each Steps in the Phase I (Candidate Selection)

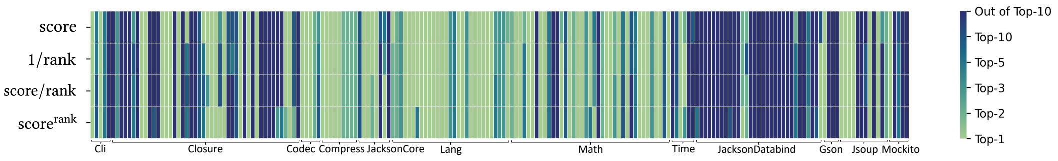


Fig. 6. Ranking Results of Different Weighting Formulas across Subjects. The horizontal axis represents all 199 subjects grouped by projects in the benchmark, and the vertical axis represents four weighting formulas. The darker the colors, the lower the BIC rankings.

5.1 RQ1. Effectiveness of BIC Identification.

In order to reduce the search space of candidate commits, SemBIC conducts commit selection in two steps initially. Figure 5 illustrates the original number of candidate commits and the number of commits remaining after selection based on loaded classes and suspicious methods respectively, depicted by median values respectively, across all bug subjects and specifically for different projects. Overall, the results confirm that our selection strategies can effectively reduce the median number of candidates to only 3.12% of the original across subjects from all projects. We observe that the method selection strategy based on suspiciousness remains effective after the candidate set has been significantly trimmed by considering only loaded classes.

To evaluate SemBIC's performance using different weighting formulas, we conduct experiments with the four formulas discussed in the previous section. Figure 6 demonstrates the ranking performance of all 199 subjects across different projects with evaluation matrix Top-k accuracy. The darker the colors, the lower the ranking of the BICs. We observe that the ranking performance of SemBIC varies across projects. SemBIC ranks BICs of most subjects from project Codec, Compress, JacksonCore, Lang, Math, and Jsoup with high accuracy, while the ranking performance in project Closure and JacksonDatabind is relatively unsatisfactory. We also find that the four weighting formulas exhibit resemblance in BIC ranking results, where the highest MRR is 0.520 when $\text{weight}(m_i) = 1/\text{rank}(m_i)$ and the lowest MRR is 0.498 when $\text{weight}(m_i) = \text{score}(m_i)^{\text{rank}(m_i)}$.

Table 2 compares the performance of BIC identification between SemBIC with weighting formula $\text{weight}(m_i) = 1/\text{rank}(m_i)$ and other ranking-based BIC identification techniques used as baselines. The results show that the ranking performance of SemBIC is better than all baselines across all evaluation metrics, except for SZZ Unleashed-RA-C with identified BFCs (i.e., SZZ-RAC*) . Specifically, SemBIC improves MRR by 13.6%, 161.1%, 417.4%, 1530.9% compared to Fonte, SZZ-RAC, Bug2Commit and FBL-BERT, respectively. It also achieves the highest accuracy across all k values in Top-k metric compared to these four baselines. Furthermore, SemBIC exhibits advantages

Table 2. Comparison on the Performance of BIC Identification.

Metric	Accuracy										MRR
	Top-1		Top-2		Top-3		Top-5		Top-10		
FBL-BERT	1	0.50%	6	3.01%	8	4.02%	10	5.02%	11	5.53%	0.032
Bug2Commit	9	4.52%	18	9.04%	21	10.55%	26	13.06%	43	21.61%	0.100
Fonte	68	34.17%	89	44.72%	105	52.76%	119	59.80%	131	65.83%	0.457
SemBIC	**88**	**44.20%**	**97**	48.74%	**112**	56.28%	**124**	62.31%	**136**	68.34%	0.520
SZZ-RAC	21	10.55%	39	19.60%	49	24.62%	58	29.15%	68	34.17%	0.199
SZZ-RAC*	21	28.77%	39	**53.42%**	49	**67.12%**	58	**79.45%**	68	**93.15%**	**0.543**

SZZ-RAC is short for SZZ Unleashed-RA-C [27].
SZZ-RAC* presents the performance of SZZ Unleashed-RA-C among the 73 subjects with BFCs identified.

in ranking the real BICs at the top, improving the Top-1 rankings for 29.4% of the bug subjects. While the SZZ algorithm demonstrates stronger BIC identification ability with BFCs, outperforming SemBIC in Top-k ($k = 2, 3, 5, 10$) ranking ratio and MRR metrics, SemBICstill ranks more bug subjects whose BICs can be ranked within Top-k ($k = 2, 3, 5, 10$) as only 73 out of total 199 bug subjects can be linked to BFCs by the SZZ algorithm. On the other hand, SemBIC still surpasses SZZ-RAC* largely in Top-1 ranking under the circumstance when BFCs are identified by the SZZ algorithm. This suggests that SZZ algorithm may suffer from low precision in the BIC identification task.

Discussion of the runtime efficiency. The time consumption for identifying BICs varies across bugs and projects, ranging from instant to hours, with a median of 2.5 minutes. Specifically, 43.2% of bugs' ranking results can be produced within 1 minute and 71.9% within 1 hour.

Answer to RQ1. The choice of weighting formulas has little effect on the BIC ranking performance of SemBIC. Overall, SemBIC can achieve good accuracy, especially in subjects from some projects. It ranks the real BICs to top-1 for 44.20% of the subjects in the evaluation benchmark. This performance surpasses that of existing techniques in general.

5.2 RQ2. Impact of FL Precision.

Since part of our approach builds upon existing FL techniques, we evaluate its effectiveness by isolating potential bottlenecks inherent in FL techniques. To achieve this, we supply perfect FL results to both SemBIC and Fonte, the latter being the only baseline that explicitly utilizes FL results, to examine how they affect the BIC ranking performance. To ensure accurate FL information, we compare the patches provided by Defects4J to locate the faults precisely and extract the buggy method using JavaParser. Figure 7 compares the top-k accuracy between these two techniques with SBFL information and with precise FL information. We can observe that SemBIC performs better as the FL results improve in all accuracy matrices, while the ranking performance of the baseline technique Fonte deteriorates with perfect FL results when $k = 1$.

Our analysis finds that the performance of Fonte is limited by the false positives arising from its heuristic where a more recent commit is considered more likely to be the BIC. For instance, the bug that triggers the failing test in subject `Lang-49` is located in the `reduce()` method within the `Fraction.java` file, part of the `org/apache/commons/lang/math` package. The code lines associated with the failing test in the buggy method were wrongly implemented when method `reduce()` was first introduced in commit `7e8976d`, which is the BIC of subject `Lang-49`. As commit `7e8976d` initially introduced the file `Fraction.java`, three other failure-associated methods with

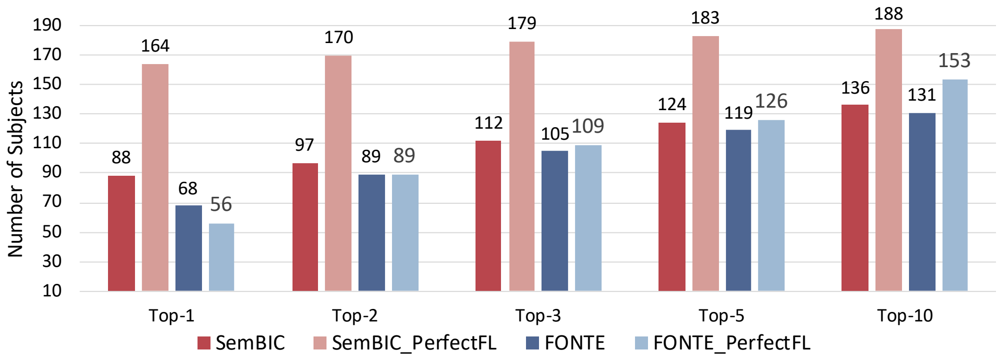


Fig. 7. Comparison on the BIC Identification Performance between SBFL and Perfect FL.

Table 3. Comparison among Utilizing Different Semantic Properties.

Accuracy	**Top-1**		**Top-2**		**Top-3**		**Top-5**		**Top-10**	
FONTE	56	28.14%	89	44.72%	109	54.77%	126	63.32%	153	76.88%
SEMBIC$_{Cons}$	152	76.38%	160	80.40%	172	86.43%	180	90.45%	188	94.47%
SEMBIC$_{Inv}$	154	77.39%	161	80.90%	175	87.94%	183	91.96%	**190**	95.48%
SEMBIC$_{Ret}$	156	78.39%	165	82.91%	177	88.94%	**184**	92.46%	189	94.97%
SEMBIC	**164**	82.41%	**170**	85.43%	**179**	89.95%	183	91.96%	188	94.47%

SEMBIC$_{Cons}$, SEMBIC$_{Inv}$ and SEMBIC$_{Ret}$ are variations of SEMBIC that uses a single semantic property. SEMBIC$_{Cons}$ uses path constraints, SEMBIC$_{Inv}$ uses method invocations and SEMBIC$_{Ret}$ uses return statement.

relatively high suspiciousness that were also modified in this commit apart from the real buggy method also count for the suspiciousness of commit 7e8976d without precise FL information. Given the buggy method, the number of suspicious methods changed by commit 7e8976d is reduced to 1 from 4. Since there are other non-BIC commits that also modified the buggy method submitted after the real BIC, FONTE's vote for the real BIC decreases based on its heuristic assumption. Thus, FONTE's BIC ranking accuracy deteriorates to Top-4 from Top-1 with precise FL information. However, SEMBIC improves the ranking performance from Top-3 to Top-1 as precise FL information removes the noisy method along the failing path, facilitating the semantic analysis of SEMBIC.

Answer to RQ2. SEMBIC performs better as the FL results improve. Therefore, we expect that SEMBIC will benefit from more precise and advanced FL techniques in the future. Surprisingly, we observe that the ranking performance of the baseline technique FONTE deteriorates with perfect FL results, potentially because of the underlying logic of method design.

5.3 RQ3. Contribution of Semantic Properties.

We further conduct an ablation study to investigate the contribution of each semantic property. To answer this question, we use each semantic property separately and compare the BIC ranking accuracy to that achieved using the complete set of semantic properties.

Table 3 presents the ranking accuracy of SEMBIC with each semantic property respectively, with FONTE serving as the baseline for comparison. Overall, SEMBIC with all semantic properties combined (i.e., SemBIC) consistently outperforms Fonte and other configurations across all evaluation metrics. Fonte has significantly lower accuracy across all metrics when compared to all variations of SemBIC. For example, in Top-1, Fonte achieves 28.14% accuracy, while the lowest among SemBIC variations is 76.38% (i.e., SemBIC$_{Cons}$), showing a substantial improvement. Each variation of SemBIC that uses a single semantic property (i.e., SemBIC$_{Cons}$, SemBIC$_{Inv}$, SemBIC$_{Ret}$) performs markedly better than Fonte, suggesting that even correctly considering a single semantic dimension can significantly enhance the ability to pinpoint BICs. Among all the individual semantic property configurations, SemBIC$_{Ret}$ performs the best in general. In some instances, SemBIC performs better with a specific semantic property (e.g., using only return statement results in more subjects having their BICs ranked in the Top-10 compared to using all semantic properties). This is because the root causes of these bugs lies in a specific semantic property. Using all semantic properties can introduce noise into the BIC identification process for these bugs.

Answer to RQ3. Semantic information plays a critical role in commit ranking for BIC identification. All the three properties contribute to the semantics summary. Our experiment results also underscore the importance of a multi-dimensional semantic analysis in enhancing the accuracy of BIC identification processes.

6 Related Work

BIC Identification. Existing techniques designed for identifying BICs can be summarized into three categories in general, i.e., dynamic, semi-dynamic and static approaches. Bisection [9] and its variants [2] are pure dynamic-based tools, requiring expensive test executions on historical commits during the binary search. Also, the dependency incompatibility issue hinders the automation of Bisection. Compared to dynamic analysis, static approaches incur lower overhead but often suffer from compromised precision. SZZ [39] and its variants [4, 11, 12, 23, 36] require the knowledge of BFCs, which might not be available at the debugging time. The main idea is to find the latest commit that shares the code changes lines with the BFC. Locus [46], Orca [3], FBL-BERT [7] and Bug2Commit [28] are IR-based methods, constituting another category of static BIC identification techniques, that heavily depend on additional information like high-quality bug reports or commit messages, potentially making them unsuitable in practice [31]. Semi-dynamic approaches enjoy the advantages of both dynamic execution and static analysis. They leverage the test runtime execution information against only one commit and analyzes the remaining commits statically. ChangeLocator [51] leverages the call stack information to identify BICs for fixed crashes based on learning a model. The SOTA technique Fonte [2] considers the commit submission time as a heuristic for identifying BICs, though it may not be effective when the heuristic is invalid. The approach SemBIC proposed by this paper is also semi-dynamic. The major difference is that SOTA Fonte is a lightweight tool that relies on predefined rules, whereas SemBIC addresses the root cause of bugs by conducting effective and practical static semantic analysis using failing path information obtained from a single execution. Moreover, there are highly relevant works that are not specifically designed for the BIC identification task. FaultLocator [54] aims to identify suspicious atomic code edits given two program versions via change impact analysis. Similarly, it uses code-level FL information to identify suspicious code edits.

Just-in-time Defective Commit Prediction. Just-in-time (JIT) defect prediction is a binary classification task. It decides whether the code changes that are defective upon the commits were submitted in order to notify developers in time for further inspection [38]. CCT5 [25] is the SOTA JIT defect prediction technique that is built on top of T5 model [32] but fine-tuned specifically for code change information. Though JIT defect prediction can provide timely help upon commit submission, the binary classification result does not contain concrete information related to the bug type or location. In other words, even when a JIT defect prediction tool identifies a defective commit, developers still lack information about the consequences of the bugs, such as which failing tests will be triggered, where the bugs are located, and when the bugs were introduced.

7 Threats to Validity

We discuss three threats that may affect our work's validity. First, the unseen cases may affect the generality of SemBIC. Our evaluation experiments are conducted with subjects in Defects4J dataset. To reduce this threat, we use subjects originally from 12 different open source projects. Since all projects are written in Java and maintained by Git, it could happen that any finding is not directly translatable to other projects, to other languages, or to projects with different characteristics. Note that we assume the test failure is caused by a bug existing in the executable code files (i.e., Java source code in our implementation). Thus, SemBIC does not generalize to bugs that are caused by non-executable files (e.g., configuration files). Second, we are concerned that the information used in IR-based baselines may affect the validity of evaluation results. As Bug2Commit and FBL-BERT rely on multiple sources of information (e.g., bug reports), the choice of information affects the comparison results. We follow the setup of these two techniques from An et al. to use bug reports as the information source, so as to ensure the evaluation fairness. Third, the accuracy of failure reproduction is crucial in failing path collection. Since the buggy commit provided by Defects4J is a version minimized according to each bug, its program behaviors during failing test execution might be slightly different from the real faulty version in the commit history. To mitigate this threat, we check the JUnit test reports generated on the real faulty commit during failing test reproduction and compare them with the test reports generated on the buggy commits provided by Defects4J to ensure that they report the same bugs.

8 Conclusion

Commits that introduce bugs into the software, known as bug-inducing commits (BICs), can provide critical information for debugging, which is challenging to identify in constantly evolving software systems. Motivated by the observation that a BIC typically introduces significant semantic changes resulting in a failing test, we propose SemBIC, a static approach to identify BIC of a bug by tracking the semantic changes associated with its failing test across a series of commits. The evaluation reveals the effectiveness of SemBIC. It ranks the actual BICs in the top position for 88 out of the 199 bugs and achieves an MRR of 0.520, surpassing the SOTA technique by 29.4% and 13.6%, respectively.

9 Data Availability

The experiment replication package of this paper is available at https://doi.org/10.5281/zenodo.14840934 [6].

Acknowledgments

We sincerely appreciate the anonymous reviewers for their insightful and constructive feedback. In addition, we thank the developers of the studied projects and the researchers of the existing works for their valuable contributions. This work is supported by the Hong Kong Research Grants Council/General Research Fund (HKSAR RGC/GRF, Grant No. 16207120) and the National Natural Science Foundation of China (Grant No. 62372193).

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*. IEEE Computer Society, 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding Bug Inducing Commits from Failures. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 589–601. https://doi.org/10.1109/ICSE48619.2023.00059

[3] Ranjita Bhagwan, Rahul Kumar, Chandra Shekhar Maddila, and Adithya Abraham Philip. 2019. Orca: Differential Bug Localization in Large-Scale Services. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association. https://www.usenix.org/conference/atc19/presentation/bhagwan

[4] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE@ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 27, 2019*, Francesca Arcelli Fontana, Bartosz Walter, Apostolos Ampatzoglou, Fabio Palomba, Gilles Perrouin, Mathieu Acher, Maxime Cordy, and Xavier Devroey (Eds.). ACM, 7–12. https://doi.org/10.1145/3340482.3342742

[5] Dylan Callaghan and Bernd Fischer. 2023. Improving Spectrum-Based Localization of Multiple Faults by Iterative Test Suite Reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1445–1457. https://doi.org/10.1145/3597926.3598148

[6] Xiao Chen. 2025. *SemBIC: Semantic-aware Identification of Bug-inducing Commits (Experiment Replication Package)*. https://doi.org/10.5281/zenodo.14840935

[7] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast Changeset-based Bug Localization with BERT. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 946–957. https://doi.org/10.1145/3510003.3510042

[8] Cobertura Contributors. 2022. Cobertura. https://cobertura.github.io/cobertura/

[9] Git Contributors. 2009. git bisect. https://git-scm.com/docs/git-bisect-lk2009

[10] Git Contributors. 2024. Git. https://git-scm.com

[11] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Trans. Software Eng.* 43, 7 (2017), 641–657. https://doi.org/10.1109/TSE.2016.2616306

[12] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *J. Softw. Evol. Process.* 26, 1 (2014), 107–139. https://doi.org/10.1002/SMR.1619

[13] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. 2016. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *CoRR* abs/1607.04347 (2016). arXiv:1607.04347 http://arxiv.org/abs/1607.04347

[14] An et al. 2022. FBL-BERT Dataset. https://anonymous.4open.science/r/fbl-bert-D567/

[15] An et al. 2023. Bug2Commit Implementation. https://github.com/coinse/fonte/blob/main/run_Bug2Commit.py

[16] The Apache Software Foundation. 2024. Ant. https://ant.apache.org/

[17] The Apache Software Foundation. 2024. Maven. https://maven.apache.org/

[18] Michael Hucka. 2018. Spiral: splitters for identifiers in source code files. *J. Open Source Softw.* 3, 24 (2018), 653. https://doi.org/10.21105/JOSS.00653

[19] JavaParser. 2019. JavaParser. https://javaparser.org

[20] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1149–1163. https://doi.org/10.1145/3372297.3417240

[21] JUnit. 2021. JUnit4. https://junit.org/junit4/

[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[23] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead Jr. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 81–90. https://doi.org/10.1109/ASE.2006.23

[24] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-Preserving Test Repair. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 217–227. https://doi.org/10.1109/ICST.2019.00030

[25] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-trained Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1509–1521. https://doi.org/10.1145/3611643.3616339

[26] Michel Maes-Bermejo, Alexander Serebrenik, Micael Gallego, Francisco Gortázar, Gregorio Robles, and Jesús M. González-Barahona. 2024. Hunting bugs: Towards an automated approach to identifying which change caused a bug through regression testing. *Empir. Softw. Eng.* 29, 3 (2024), 66. https://doi.org/10.1007/S10664-024-10479-Z

[27] Michel Maes-Bermejo, Alexander Serebrenik, Micael Gallego, Francisco Gortázar, Gregorio Robles, and Jesús María González Barahona. 2024. Hunting bugs: Towards an automated approach to identifying which change caused a bug through regression testing. *Empirical Software Engineering* 29, 3 (2024), 66.

[28] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2021. Industry-Scale IR-Based Bug Localization: A Perspective from Facebook. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 188–197. https://doi.org/10.1109/ICSE-SEIP52600.2021.00028

[29] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Inf. Syst.* 56 (2016), 157–173. https://doi.org/10.1016/J.IS.2015.08.004

[30] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 609–620. https://doi.org/10.1109/ICSE.2017.62

[31] Sophia Quach, Maxime Lamothe, Yasutaka Kamei, and Weiyi Shang. 2021. An empirical study on the use of SZZ for identifying inducing changes of non-functional bugs. *Empir. Softw. Eng.* 26, 4 (2021), 71. https://doi.org/10.1007/S10664-021-09970-8

[32] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[33] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 432–448.

[34] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. 1995. Large Test Collection Experiments on an Operational, Interactive System: Okapi at TREC. *Inf. Process. Manag.* 31, 3 (1995), 345–360. https://doi.org/10.1016/0306-4573(94)00051-4

[35] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesús M. González-Barahona. 2020. How bugs are born: a model to identify how bugs are introduced in software components. *Empir. Softw. Eng.* 25, 2 (2020), 1294–1340. https://doi.org/10.1007/S10664-019-09781-Y

[36] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2023. A comprehensive evaluation of SZZ Variants through a developer-informed oracle. *J. Syst. Softw.* 202 (2023), 111729. https://doi.org/10.1016/J.JSS.2023.111729

[37] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*. 1157–1168.

[38] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 62. https://doi.org/10.1145/2393596.2393670

[39] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM. https://doi.org/10.1145/1083142.1083147

[40] soot oss. 2012. Soot Jimple Expressions. https://soot-oss.github.io/soot/docs/4.3.0/jdoc/soot/jimple/package-summary.html

[41] soot oss. 2022. Soot. https://soot-oss.github.io/soot/

[42] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and Named Entity Recognition in StackOverflow. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 4913–4926. https://doi.org/10.18653/V1/2020.ACL-MAIN.443

[43] Raja Vallée-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations. https://api.semanticscholar.org/CorpusID:10529361

[44] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Trans. Software Eng.* 47, 11 (2021), 2348–2368. https://doi.org/10.1109/TSE.2019.2948158

[45] Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2020. Boosting automated program repair with bug-inducing commits. In *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 77–80. https://doi.org/10.1145/3377816.3381743

[46] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 262–273. https://doi.org/10.1145/2970276.2970359

[47] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 326–337. https://doi.org/10.1145/3338906.3338962

[48] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 87–98.

[49] Wikiversity. 2024. Vector Space Model. https://en.wikipedia.org/wiki/Vector_space_model

[50] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[51] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: locate crash-inducing changes based on crash reports. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 536. https://doi.org/10.1145/3180155.3182516

[52] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. 2013. Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation. In *Search Based Software Engineering - 5th International Symposium, SSBSE 2013, St. Petersburg, Russia, August 24-26, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8084)*, Günther Ruhe and Yuanyuan Zhang (Eds.). Springer, 224–238. https://doi.org/10.1007/978-3-642-39742-4_17

[53] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 347–359. https://doi.org/10.1145/3597926.3598061

[54] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 23–32. https://doi.org/10.1109/ICSM.2011.6080769

Received 2024-09-13; accepted 2025-01-14