FONTE: Finding Bug Inducing Commits from Failures

Gabin An
School of Computing, KAIST
Daejeon, Republic of Korea
agb94@kaist.ac.kr

Jingun Hong
SAP Labs Korea
Seoul, Republic of Korea
jingun.hong@sap.com

Naryeong Kim
School of Computing, KAIST
Daejeon, Republic of Korea
kimnal1234@kaist.ac.kr

Shin Yoo
School of Computing, KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

arXiv:2212.06376v1 [cs.SE] 13 Dec 2022

***Abstract*—A Bug Inducing Commit (BIC) is a commit that introduces a software bug into the codebase. Knowing the relevant BIC for a given bug can provide valuable information for debugging as well as bug triaging. However, existing BIC identification techniques are either too expensive (because they require the failing tests to be executed against previous versions for bisection) or inapplicable at the debugging time (because they require post hoc artefacts such as bug reports or bug fixes). We propose FONTE, an efficient and accurate BIC identification technique that only requires test coverage. FONTE combines Fault Localisation (FL) with BIC identification and ranks commits based on the suspiciousness of the code elements that they modified. FONTE reduces the search space of BICs using failure coverage as well as a filter that detects commits that are merely style changes. Our empirical evaluation using 130 real-world BICs shows that FONTE significantly outperforms state-of-the-art BIC identification techniques based on Information Retrieval as well as neural code embedding models, achieving at least 39% higher MRR. We also report that the ranking scores produced by FONTE can be used to perform weighted bisection, further reducing the cost of BIC identification. Finally, we apply FONTE to a large-scale industry project with over 10M lines of code, and show that it can rank the actual BIC within the top five commits for 87% of the studied real batch-testing failures, and save the BIC inspection cost by 32% on average.**

***Index Terms*—Bug Inducing Commit, Fault Localisation, Git, Weighted Bisection, Batch Testing**

I. INTRODUCTION

A Bug Inducing Commit (BIC) [1] refers to a commit that introduces buggy source code into the program. Accurate identification of BICs can have many benefits. Existing work has shown that simply reverting BICs may suffice for bug fixes [2], [3], while the knowledge of BICs can aid manual debugging by developers [4]. Considering the finding that 78% of bugs are fixed by those who introduced them in the first place [5], the knowledge of BICs can help effective assignment of a newly revealed bug to the right team or developers [6]. Finally, for software engineering researchers, a BIC dataset can be used to study how bugs are created [1], [7], eventually resulting in better software defect prediction techniques [8].

Many approaches have been proposed to identify BICs: existing approaches can be categorised into three groups. The first group is bisection, i.e., a binary search on the commit history that checks whether each snapshot in the commit history is buggy or not [9]. The actual inspection can be done either manually, or automatically by executing the bug-revealing test cases. While the binary search itself is efficient, the inspection cost required for each snapshot can render bisection impractical. For example, with large-scale software projects, the cost of simply building and executing test cases for a specific snapshot can be significantly high.

While bisection depends on explicitly checking each snapshot, other approaches are static, i.e., they only concern commit histories or bug reports. The second group is represented by SZZ [1] and its variants. Given a Bug Fixing Commit (BFC), SZZ essentially seeks to identify a set of commits that last modified each element of BFC. However, SZZ-like approaches require BFCs as input, which are only available when the bug has already been patched. Consequently, these techniques are not applicable at the debugging time. The third group is IR-based BIC identification [5], [10], which reformulates BIC identification as Information Retrieval (IR) where the bug report is the query, and the commits are the documents. Given a query, i.e., the bug report that contains various information about the failure in question, the BIC is likely to be the commit that is the most lexically similar to the query. While IR-based approaches do not incur the cost of their dynamic counterparts (e.g., compilation and test execution), they cannot be applied if a bug report for the latest failure is not available yet, or if it does but its quality is too low.

In this paper, we aim to propose a BIC identification technique that is accurate, efficient, and available at debugging time. Intuitively, our technique, FONTE[1], distributes the suspiciousness computed by a Fault Localisation (FL) technique for the current bug to commits in the development history, expanding the dimensions of FL techniques from the location within the codebase (i.e., spatial) to the history of the codebase (i.e., temporal). FONTE starts by collecting the test coverage at the time of failure and computing suspiciousness scores for code elements using a FL technique [11]. Subsequently, FONTE traces back the commits that are relevant to the code covered by failing tests [12]. Commits that are mere style changes are filtered out based on Abstract Syntax Tree (AST) level comparisons. Finally, the remaining commits are ranked according to the suspiciousness of the current code that is modified by each candidate commit. Compared to bisection, FONTE does not require inspection of each snapshot

[1]Fonte is an Italian word meaning “source” or “origin”.

it considers and therefore more efficient. Instead, FONTE uses the failure coverage to improve accuracy. It also does not require bug reports or bug fixes, and therefore can be applied at debugging time once a test failure is observed.

We evaluate FONTE with a benchmark of 130 real-world bugs: 67 from an existing BIC dataset, and 63 that are manually curated by us. The results show that the ranking of BICs produced by FONTE achieves 242% of MRR compared to the random baseline. Furthermore, exploiting the fact that FONTE assigns scores to each candidate BIC, we also propose a weighted bisection method that leverages the commit scores during the search. Weighted bisection combined with FONTE can save the number of bisection iterations for 98% of the cases. Since FONTE does not require any manual human effort, it can be easily incorporated into CI pipelines to provide developers with candidate BICs when reporting test failures.

The contributions of this paper are summarised as follows:

- We present FONTE, a BIC identification technique that only requires information from the failed test execution and the commit history. Since it does not require bug patches or bug reports, FONTE can be used to aid debugging by providing developers with the relevant BICs, once a test failure is observed.
- We evaluate FONTE with 130 real-world bugs and show that FONTE can accurately rank BIC candidates: it achieves MRR of 0.528, which outperforms a state-of-the-art IR-based BIC identification technique by 39%.
- We introduce weighted bisection that uses the scores assigned to candidate BICs by FONTE. Weighted bisection can reduce the required number of iterations for about 98% of studied bugs when compared to the standard bisection algorithm applied to the entire commit history.
- We apply FONTE to the batch testing scenario of large-scale industry software. It achieves 547% of MRR compared to the random baseline and can reduce the bisection iterations in 78% of cases.
- We release a new BIC benchmark dataset for 130 Defects4J version 2.0 bugs. FONTE is publicly available at https://figshare.com/s/22e74b7bdf675e499ade, along with artefacts of the empirical evaluation in this paper.

The remainder of the paper is structured as follows. Section II explains the research context of this paper and defines the basic notations. Section III and IV propose FONTE and the novel weighted bisection method, respectively. Section V describes the empirical evaluation settings for FONTE along with the research questions, and Section VI presents the results. Section VII shows the application results of FONTE to the batch testing scenario in industry software. Section VIII addresses the threats to validity, and Section IX covers the related work of FONTE. Finally, Section X concludes.

II. BACKGROUND

This section provides the background of this paper.

A. Research Context

Debugging is usually initiated by observing a failure that reveals a bug in the program. Even when a field failure is reported by users, the debugging activities typically start with reproducing the field failure [13]–[15]: this is because failure-triggering test cases are essential to confirm whether the bug is fixed or not. Once observed, the test failure goes through the bug triage phase to be assigned to a developer or a team, who will analyse the buggy behaviour and produce a patch. Knowing the BIC responsible for the observed failure can not only contribute to more efficient bug triage [6] but also help developers better understand the context of the buggy behaviour [5]. To identify BICs as soon as the bug is detected, we cannot rely on any information that is produced later in the debugging process, such as bug fixes or bug reports.

While some BIC identification techniques [6], [10] are based on the information from failures, they only use the stack traces or the exception messages, which may only be indirectly linked to the contents of actual BICs. Given that commits are directly coupled to specific locations in the source code, we focus on the actual coverage of the failing tests as the main source of information. Existing work shows that the coverage of failing test executions (i.e., *failure coverage*) can reduce the BIC search space very effectively [12]: simply by filtering out any commit that is not related to the evolution of code elements covered by the failing tests, the search spaces of BICs for 703 bugs in Defects4J [16] were reduced to 12.4% of their original size on average. The high reduction rate suggests that failure coverage has the potential to provide the basis for a BIC identification technique available at the debugging time.

In this work, we aim to accurately locate the BIC using only the information that is available at the debugging time, just after the observation of test failure. We build upon the previous technique of BIC search space reduction [12], and present a technique that can accurately measure the relevance of each commit in the reduced search space to the failure. Intuitively, our approach distributes the code level suspiciousness measured in the current buggy version to the past commits.

B. Basic Notations

Let us define the following properties of a program P:

- A set of commits $C = \{c_1, c_2, \ldots\}$ made to P
- A set of code elements $E = \{e_1, e_2, \ldots\}$ of P, such as statements or methods
- A set of test cases $T = \{t_1, t_2, \ldots\}$ where $T_F \subseteq T$ is a set of failing test cases

We assume that the bug responsible to the failing tests resides in the source code, i.e., some elements in E cause the failure of T_F. We also define the following relations on sets C, T, and E:

- A relation $\mathsf{Cover} \subseteq T \times E$ defines the relation between test cases and code elements. For every $t \in T$ and $e \in E$, $(t, e) \in \mathsf{Cover}$ if and only if the test t covers e during the execution.

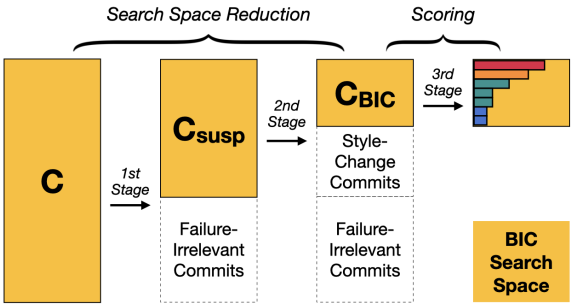

Fig. 1. Overview of FONTE

- A relation Evolve $\subseteq C \times E$ defines the relation between commits and code elements. For every $c \in C$ and $e \in E$, $(c, e) \in$ Evolve if and only if the commit c is in the change history of the code element e.

As our ultimate goal is to find the BIC in C, we aim to design a scoring function $s\colon C \to \mathbb{R}$ that gives higher scores to commits that have a higher probability of being the BIC.

III. FONTE: AUTOMATED BIC IDENTIFICATION VIA DYNAMIC, SYNTACTIC, AND HISTORICAL ANALYSIS

This paper presents FONTE, a technique to automatically identify the BIC, based on the assumption that *a commit is more likely to be a bug inducing commit if it introduced or modified a code element that is more relevant to the observed failure*. The key idea behind FONTE is that the relevancy of the code elements to the observed failures can be quantified using existing FL techniques [11], such as SBFL. FONTE distributes the code-level suspiciousness, computed by an FL technique for the observed failures, to commits in the development history. Fig. 1 illustrates the three stages of FONTE, which are described below:

1) FONTE identifies all suspicious code elements using the coverage of failing test cases and discards the commits that are irrelevant to the code elements from the BIC search space [12].
2) FONTE additionally filters out the commits that contain only style changes to the suspicious files using AST level comparisons.
3) FONTE assigns scores to the remaining commits in the search space using the FL scores and evolution history of the suspicious code elements.

The rest of this section describes each stage in more detail.

A. Stage 1: Filtering Out Failure-Irrelevant Commits

Using the notations defined in Section II-B, we can represent the failure-coverage-based BIC search space reduction [12] as follows. First, let $E_{susp} \subseteq E$ denote the set of all code elements that are covered by the failing test cases:

$$E_{susp} = \bigcup_{t \in T_F} \{e \in E | (t, e) \in \mathsf{Cover}\} \tag{1}$$

```
1 @@ -74,7 +74,7 @@ import org.apache.commons.lang.
      exception.NestableRuntimeException;
2   * @author Phil Steitz
3   * @author Pete Gieser
4   * @since 2.0
5 - * @version $Id: StringEscapeUtils.java,v 1.26
      2003/09/07 14:32:34 psteitz Exp $
6 + * @version $Id: StringEscapeUtils.java,v 1.27
      2003/09/13 03:23:24 psteitz Exp $
7   */
8  public class StringEscapeUtils {
9
10 @@ -242,7 +242,9 @@ public class StringEscapeUtils {
11              } else {
12                  switch (ch) {
13                      case '\'':
14 -                        if (escapeSingleQuote) out.write
      ('\\');
15 +                        if (escapeSingleQuote) {
16 +                          out.write('\\');
17 +                        }
18                          out.write('\'');
19                          break;
20                      case '"':
```

Fig. 2. Changes by the commit `5814f50` in Defects4J `Lang-46`

Subsequently, we obtain $C_{susp} \subseteq C$, a set of commits that are involved in the evolution of at least one code element in E_{susp}:

$$C_{susp} = \bigcup_{e \in E_{susp}} \{c \in C | (c, e) \in \mathsf{Evolve}\} \tag{2}$$

Then, all commits not contained in C_{susp} can be discarded from our BIC search space because the changes introduced by those commits are not related to any code element executed by failing executions. Consequently, the BIC search space is reduced from C to C_{susp}.

B. Stage 2: Filtering Out Style Change Commits

The reduced set of candidate BICs, C_{susp}, may still contain *style change commits*, i.e., commits that do not introduce any semantic change to the suspicious code elements. These commits can be further excluded from the BIC search space, as they cannot have altered the functional behaviour of the relevant code elements [17]. An example of such a commit is shown in Fig. 2, which modifies the comments and encloses the single statement in the `if` block.

We use the AST level comparison [18] to identify whether a given commit $c \in C_{susp}$ is a style change commit or not. First, we identify the set of files, F, that are modified by the commit c and covered by the failing test cases. Formally, any file in F contains at least one code element in:

$$E^c_{susp} = \{e \in E_{susp} | (c, e) \in \mathsf{Evolve}\} \tag{3}$$

Then, for each file $f \in F$, we compare the ASTs derived from f before and after the commit c. If the ASTs are identical for all files in F, we consider the commit c as a style-change commit. Note that this approach does not guarantee 100% recall, as it is possible for two source files to yield different ASTs while sharing the same semantic. However, it can safely prune the search space due to its soundness, i.e., if it identifies

TABLE I
EXAMPLE OF THE VOTING POWER OF CODE ELEMENTS

Code Element		e_1	e_2	e_3	e_4	e_5
Score		1.0	0.6	0.6	0.6	0.3
$rank_{max}$		1	4	4	4	5
$rank_{dense}$		1	2	2	2	3
$vote$	$\alpha = 0, \tau = max$	1.00	0.25	0.25	0.25	0.20
	$\alpha = 1, \tau = max$	1.00	0.15	0.15	0.15	0.06
	$\alpha = 0, \tau = dense$	1.00	0.50	0.50	0.50	0.33
	$\alpha = 1, \tau = dense$	1.00	0.30	0.30	0.30	0.10

a commit as a style change commit, it is guaranteed to be a style change commit. Consequently, the search space for BIC can be further reduced to $C_{BIC} = C_{susp} \setminus C_{sc}$, in which C_{sc} denotes all identified style change commits in C_{susp}.

C. Stage 3: Scoring Commits using FL Scores and History

We are now left with the reduced BIC search space C_{BIC}, which only contains commits related to the evolution of the suspicious code elements and are also identified as non-style-change commits. The remaining task is to rank the commits in C_{BIC} in the order of their likelihood of being the BIC. As mentioned earlier, our basic intuition is that if a commit had created, or modified, more suspicious code elements for the observed failures, it is more likely to be a BIC.

The suspiciousness of code elements can be quantified via an FL technique. For example, we can apply SBFL [11] using the coverage of the test suite T: note that SBFL uses only test coverage and result information, both of which are available at the time of observing a test failure. Assuming that we are given the suspiciousness scores, let $susp$: $E_{susp} \rightarrow [0, \infty)$ be the mapping function from each suspicious code element in E_{susp} to its non-negative FL score.[2] To convert the code-level scores to the commit level, we propose a voting-based commit scoring model where the FL score of a code element is distributed to its relevant commits. The model has two main components: rank-based voting power and depth-based decay.

Rank-based Voting Power: Recent work [19]–[21] showed that, when aggregating FL scores from finer granularity elements (e.g, statements) to a coarser level (e.g., methods), it is better to use the *relative rankings* from the original level only, rather than directly using the scores. The actual aggregation takes the form of voting: the higher the ranking of a code element is in the original level, the more votes it is assigned with for the target level. Subsequently, each code element casts its votes to the related elements in the target level. We adopt this voting-based method to aggregate the statement level FL scores to commits. The *voting power* of each code element e based on their FL rankings (and scores) as follows:

$$vote(e) = \frac{\alpha * susp(e) + (1 - \alpha) * 1}{rank_\tau(e)} \quad (4)$$

[2]The constraint of FL-score being non-negative is adopted for the sake of simplicity. Note that any FL results can be easily transformed so that the lowest score is 0.

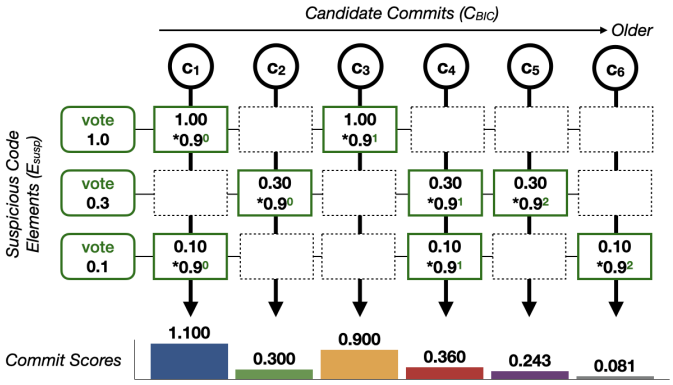

Fig. 3. Example of computing the commit scores when $\lambda = 0.1$

where $\alpha \in \{0, 1\}$ is a hyperparameter that decides whether to use the suspiciousness value ($\alpha = 1$) as a numerator or not ($\alpha = 0$), and τ a hyperparameter that defines the tie-breaking scheme. We vary $\tau \in \{max, dense\}$: the max tie-breaking scheme gives the lowest (worst) rank in the tied group to all tied elements, while $dense$ gives the highest but does not skip any ranks after ties. By design, $\tau = max$ will penalise tied elements more severely than $\tau = dense$. The example in Table I shows how the hyperparameters affect voting. Note that the relative order between FL scores is preserved in the voting power regardless of hyperparameters, i.e., $vote(e) > vote(e')$ if and only if $susp(e) > susp(e')$.

Depth-based Decay: Wen et al. [5] showed that using the information about commit time can boost the accuracy of the BIC identification. Similarly, Wu et al. [2] observed that the commit time of crash-inducing changes is closer to the reporting time of the crashes. Based on those findings, we hypothesise that older commits are less likely to be responsible for the currently observed failure, because if an older commit was a BIC, it is more likely that the resulting bug has already been found and fixed. To capture this intuition, we propose a depth-based decay function that decreases the voting power of a code element as the depth of the commit in the history of the code element increases. The historical depth of a commit c, with respect to a code element $e \in E^c_{susp}$ (Eq. 3), is defined as follows:

$$\begin{aligned} depth(e, c) = &|\{c' \in C_{BIC}| \\ &(c', e) \in \mathsf{Evolve} \wedge c'.time > c.time\}| \end{aligned} \quad (5)$$

Note that, unlike existing work that considered the depth at the level of commit, we consider the depth of each code element and use this to adjust the voting power of each element.

Bringing it all together (Eq. 4 and Eq. 5), we use the following model to assign a score to each commit c in C_{BIC}:

$$commitScore(c) = \sum_{e \in E^c_{susp}} vote(e) * (1 - \lambda)^{depth(e,c)} \quad (6)$$

where $\lambda \in [0, 1)$ is the decay factor: when $\lambda = 0$, there is no penalty for older commits. Figure 3 shows the example of calculating the score of commits when $\lambda = 0.1$.

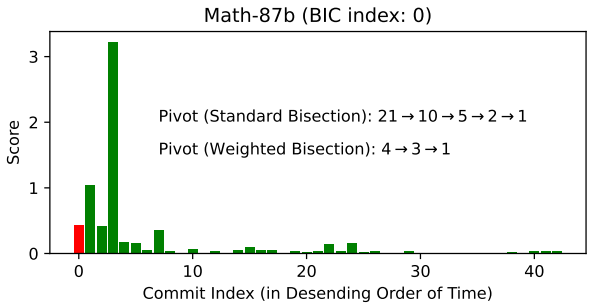

Fig. 4. Example of applying the weighted bisection to `Math-87`

Finally, based on $commitScore$, the commit scoring function $s\colon C \to [0, \infty)$ of FONTE is defined as follows:

$$s(c) = \begin{cases} commitScore(c) & \text{if } c \in C_{BIC} \\ 0 & \text{otherwise} \end{cases}$$

IV. Weighted Bisection

Bisection is a traditional way of finding the BIC by repeatedly narrowing down the search space in half using binary search: it is implemented in popular Version Control Systems (VSCs), e.g., `git bisect` or `svn-bisect`. A standard bisection is performed as follows: given the earliest *bad* and last *good* versions, it iteratively checks whether the midpoint of those two versions, referred to as a *pivot*, contains the bug. If there is a bug, the earliest bad point is updated to the pivot, otherwise, the last good point is updated to the pivot. If there is a bug-revealing test case that can automatically check the existence of a bug, the search process can be fully automated.

However, as pointed out in previous work [6], even though the bug existence check can be automated, each bisect iteration may still require a significant amount of time and computing resources, especially when the program is large and complex, or the bug-revealing test takes a long time to execute. Since a lengthy bisection process can block the entire debugging pipeline, we aim to explore whether the bisection can be accelerated using the commit score information.

We propose a *weighted* bisection algorithm, where the search pivot is set to a commit that will halve *the amount of remaining commit scores* instead of *the number of remaining commits*, in order to reduce the number of highly suspicious commits more quickly. For example, let us consider the example in Fig. 4 that shows the score distribution of the commits in the reduced BIC search space of `Math-87` in Defects4J. For `Math-87`, the score distribution is biased towards a small number of recent commits including the real BIC (marked in red) with the third highest score. In this case, simply using the midpoint as a search pivot might not be a good choice because all highly suspicious commits still remain together on one side of the split search space: as a result, the standard bisection requires five iterations to finish. Alternatively, if we pivot at the commit that halves the amount of remaining scores, the bisection reaches the actual BIC more quickly, completing the search in three iterations. Note that this algorithm is a *generalised* version of the standard bisection: the standard bisection is a special case of weighted bisection with all commits sharing the same non-zero score.

Algorithm 1: Weighted Bisection Algorithm

Input: Set of commits C
Input: Commit score (weight) function $s : C \to [0, \infty)$
Output: Bug inducing commit $c \in C$

```
// Remove irrelevant commits
1  C ← {c ∈ C | s(c) > 0}
2  C' ← C.orderByTimeDesc()
   // C'[i] is newer than C'[i+1]
3  bad ← 0
4  good ← |C'|
5  while good > bad + 1 do
       // S(a,b) = Σ_{i=a}^{b} s(C'[i])
6      pivot ← argmin_{i=bad+1}^{good-1} |S(bad, i-1) − S(i, good-1)|
7      if C'[pivot].ContainsBug() then
8          bad ← pivot
9      else
10         good ← pivot
   // assert bad = good − 1
11 return C'[bad]
```

Algorithm 1 presents the weighted bisection algorithm. It takes the set of commits C, and the commit score function $s \in C \to [0, \infty)$, as input, and returns the BIC. First, it removes all commits with a score of zero (Line 1), and sorts the remaining commits in the descending order of their commit time (Line 2). Assuming that there is at least one BIC in the sorted sequence C', the earliest bad index bad is set to 0, the index of the most recent commit (Line 3). Since all commits in C' are BIC candidates, we set the last good index $good$ to the index just after the oldest commit (Line 4). Then, a new $pivot$ index is iteratively selected from the range $[bad+1, good-1]$, until there is no remaining commit between bad and $good$ (Line 5). As mentioned earlier, we select a pivot that minimises the difference between the left (not including pivot) and the right (including pivot) sum of the scores (Line 6). Once a new pivot is selected, the commit $C'[pivot]$ is inspected for the bug, either using the bug-revealing tests or manually (Line 7). If a bug is detected, the bad index is updated to $pivot$ (Line 8), otherwise, the $good$ index is updated to $pivot$ (Line 10). Finally, it returns the identified BIC at the bad index when the loop terminates (Line 11).

V. Evaluation Setup

This section presents our research questions and describes the experimental setup.

A. Research Questions

We ask the following research questions in this paper:

- **RQ1**: How accurately does FONTE rank the BIC?
- **RQ2**: How efficient is the weighted bisection compared to the standard bisection?
- **RQ3**: What is the impact of FL accuracy to the performance of FONTE?

TABLE II
EXAMPLE OF RELEVANT TEST SELECTION (TIME-15)

Failing Test (T_F)
org.joda.time.field.TestFieldUtils::testSafeMultiplyLongInt
Classes Covered by the Failing Test
org.joda.time.field.**FieldUtils** org.joda.time.**IllegalFieldValueException**
Relevant Tests ($T \setminus T_F$)
org.joda.time.Test**IllegalFieldValueException**::testGJCutover org.joda.time.Test**IllegalFieldValueException**::testJulianYearZero org.joda.time.Test**IllegalFieldValueException**::testOtherConstructors org.joda.time.Test**IllegalFieldValueException**::testReadablePartialValidate org.joda.time.Test**IllegalFieldValueException**::testSetText org.joda.time.Test**IllegalFieldValueException**::testSkipDateTimeField org.joda.time.Test**IllegalFieldValueException**::testVerifyValueBounds org.joda.time.Test**IllegalFieldValueException**::testZoneTransition org.joda.time.field.Test**FieldUtils**::testSafeAddInt org.joda.time.field.Test**FieldUtils**::testSafeAddLong org.joda.time.field.Test**FieldUtils**::testSafeMultiplyLongLong org.joda.time.field.Test**FieldUtils**::testSafeSubtractLong

B. Dataset of Bug Inducing Commits

We choose Defects4J v2.0.0 [16], a collection of 835 real-world bugs in Java open-source programs, as the source of our experimental subjects. While Defects4J provides test suites containing the bug-revealing tests for every bug, as well as the entire commit history for each buggy version, it lacks the BIC information for each bug. We, therefore, start with a readily-available BIC dataset for 91 Defects4J bugs[3] constructed by Wen et al. [4]. This dataset was created by running the bug-revealing test cases on the past versions and finding the earliest buggy version that makes the tests fail. However, in our experiment, we are forced to exclude 24 out of 91 data points. Since FONTE is implemented using Git, it cannot trace the commit history of nine bugs from the `JFreeChart` project which uses SVN as its version control system. Further, we exclude 14 data points that are shown to be inaccurate by previous work [12]. Lastly, `Time-23` is also discarded, because we found that the identified commit in the dataset does not contain any change to code, but only to the license comments. The detailed reasons can be found in our repository.

We augment the remaining 67 BIC data points from Wen et al. [4] by inspecting some of the remaining bugs in Defects4J. Two of the authors manually and independently identified the BIC for each bug, consulting the bug reports, failure symptoms, and developer patches. To reduce the manual inspection cost, we only targeted the bugs for which the cardinality of the reduced BIC search space, C_{BIC}, is less than or equal to ten. Two authors reached on consensus for 70 data points, which have been added to the dataset. In summary, a total of 130 data points (67 from Wen et al. [4] + 70 manually curated - 7 overlapped) are used for the evaluation of FONTE. The combined BIC dataset and the provenance of each data point are available in our repository for further scrutiny.

C. Implementation Details

We apply FONTE at the *statement*-level granularity, i.e., E is a set of statements composing the target buggy program. The initial BIC search space, C, is set to all commits from the very first commit up to the commit correspond to the buggy version, i.e., `revision.id.buggy` in Defects4J. Among the given test suite in Defects4J, we only use the bug-revealing (i.e., failing) test cases as well as their relevant test cases as T. A test case is relevant if and only if its full name contains the name of at least one class executed by the failing test cases. Table II shows the example of the relevant test selection.

1) Construction of the Cover *relation:* To construct the Cover relation between T and E, we measure the statement-level coverage of each test case in T using `Cobertura v2.0.3` which is included in Defects4J.

2) Construction of the Evolve *relation:* To construct the Evolve relation between C and E, we need to retrieve the commit history of each code element: we use the `git log` command[4] following An et al. [12]. We also attempted using CodeShovel [22], a state-of-the-art method history retrieval tool, instead of `git log`, but found that the tool sometimes produces incorrect histories. Since it is infeasible to manually validate all commits retrieved by CodeShovel, we only report the results with `git log` in this paper. However, we include the commit history retrieved by CodeShovel (with the incorrect outputs) in our artefact for further validation and comparison.

Please note that for each statement, we retrieve the commit history of its enclosing method and create the Evolve relations between the statements and the retrieved commits to ensure high recall for commit histories. This is also to deal with omission bugs [23]: if a bug is caused by omission of some statements, we cannot trace the log of the missing statements because they literally do not exist in the current version. In that case, tracing the log of the neighbouring statements (in the enclosing method) will enable to find the inducing commit, as the method that encloses the omission bug should have been covered by the failing tests [12].

3) Detection of Style-Change Commits: For Stage 2, we use OpenRewrite v7.21.0[5] to ensure the same coding standard between the two versions of files. More specifically, we use the `Cleanup` recipe[6] that fixes any errors that violate CheckStyle rules.[7] This ensures that trivial differences between two versions that do not lead to semantic differences are ignored: a good example is a commit in `Lang`, which is shown in Fig. 2. To compare ASTs, we use the isomorphism test of GumTree v3.0.0 [18] that has time complexity of $O(1)$.

4) Fault Localisation: To obtain the FL score of each statement, we use a widely-used SBFL formula, Ochiai [24],

[3]https://github.com/justinwm/InduceBenchmark

[4]`git log -C -M -L<start_line>,<end_line>:<file>`. The options `-C` and `-M` detect file rename/copy/move between versions.

[5]https://github.com/openrewrite/rewrite

[6]https://docs.openrewrite.org/reference/recipes/java/cleanup

[7]https://checkstyle.sourceforge.io/

which can be expressed in our context as follows:

$$Ochiai(e) = \frac{|\{t \in T_F | (t,e) \in \mathsf{Cover}\}|}{\sqrt{|T_F| * |\{t \in T | (t,e) \in \mathsf{Cover}\}|}}$$

By definition, $Ochiai(e) > 0$ if and only if $e \in E_{susp}$ (Eq. 1).

D. Evaluation Metrics

If the scoring model works well, BICs will have higher scores and ranks than other commits. Therefore, we use the following widely-adopted ranking-based evaluation metrics. When there are tied elements, the max-tiebreaker is used.

- Accuracy@n: The number of subjects where the ranking of the BIC is within the top n positions (*higher is better*)
- Mean Reciprocal Rank (MRR) [25]: The average reciprocal rank of the BIC (*higher is better*)

E. Baselines

We compare FONTE to the following baselines.

1) Other Voting Schemes:

- Equal: All lines that are covered by failing test cases are assigned the same weight, i.e., $vote(e) = 1$ (7).
- Only Score: The voting power of a method is simply defined as its FL score without considering the ranking, i.e., $vote(e) = susp(e)$ (8).

2) Other Scoring/Ranking Techniques:

- Random: This strategy randomly shuffles the commits in the search space. The random strategy is a meaningful baseline for ranking-based evaluation because the ranking result can be overestimated when the size of the search space is small. When there are n commits in the search space, the expected rank of the BIC is $\frac{1+n}{2}$.
- Max (Eq. 9): Instead of Eq. 6, the score of a commit is defined as the highest suspiciousness score of code elements that are modified by the commit:

$$commitScore(c) = \max_{e \in E^c_{susp}} susp(e) \quad (9)$$

  Similarly, in Orca [10], the file-level scores are converted into the commit level using max-aggregation. Many FL techniques have used this scheme when the granularity of the code elements in the coverage matrix and the target FL granularity are different [26], [27].
- FBL-BERT [28]: FBL-BERT is a recently proposed changeset localisation technique based on a pre-trained BERT model called BERTOverflow [29]. Given a bug report, it retrieves the relevant changesets using their scores obtained by the BERT-based model. We fine-tune the model using the training dataset from the `JDT` project, which is the largest training dataset provided by their repository[8]: this is because no such training data is available for our target projects. We use the ARC changeset encoding strategy, which performed the best for changeset-level retrieval in the original study [28]. As Defects4J contains the link to the original bug report for every bug, we use the contents of the original bug report as an input query.
- Bug2Commit [6]: Bug2Commit is a state-of-the-art IR-based BIC identification method for large-scale systems: it exploits multiple features of commits and bug reports. When implementing Bug2Commit, we use the Vector Space Model (VSM) because the word-embedding model requires an additional training dataset of bug reports and commits. As in the original paper, we use BM25 [30] as a vectoriser and use the Ronin tokeniser, the most advanced splitter in `Spiral` [31][9]. We use two features of commit: the commit message and the modified file names. From bug reports, we use three features: the exception message and stack traces from failing test cases, the title of bug report, and the content of bug report.

[8]We confirm that the model fine-tuned with `JDT` performs better than that fine-tuned with `ZXing`, which has the smallest training dataset.

VI. RESULTS

This section presents the results of our experiments.

A. *RQ1*: Ranking Performance of FONTE

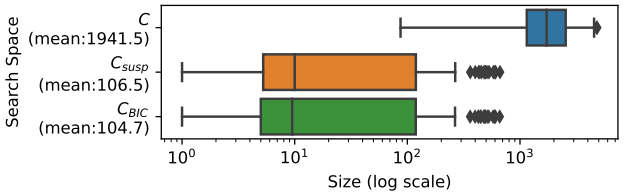

Fig. 5. Distributions of the sizes of search space

TABLE III
THE DISTRIBUTION OF THE SIZE OF THE REDUCED SEARCH SPACE

$\lvert C_{BIC}\rvert$	≤ 1	≤ 2	≤ 3	≤ 5	≤ 10	≤ 20	≤ 30
# Subjects	3	8	19	41	71	76	76
$\lvert C_{BIC}\rvert$	≤ 50	≤ 100	≤ 200	≤ 300	≤ 500	≤ 600	≤ 700
# Subjects	83	94	106	112	123	128	130

Let us first check how much search space reduction is achieved by Stages 1 and 2 of FONTE. Figure 5 shows the distribution of the sizes of C (original), C_{susp} (after Stage 1), and C_{BIC} (after Stage 1+2), respectively, over all subjects. The results show that Stage 1 significantly reduces the size of the search space. On average, the size of search space can be reduced to 7% of its original size using only the coverage of the failing tests. The search space reduction by Stage 2 is relatively marginal compared to Stage 1: the style change commits are detected in only 75 out of 130 subjects (58%). To make it easier for readers to grasp the ranking-based evaluation results that follow, we report the size distribution of the final reduced search space C_{BIC} in Table III.

We now turn to the ranking performance of FONTE: Fig. 6 presents the MRR metric achieved by FONTE with each

[9]https://github.com/casics/spiral

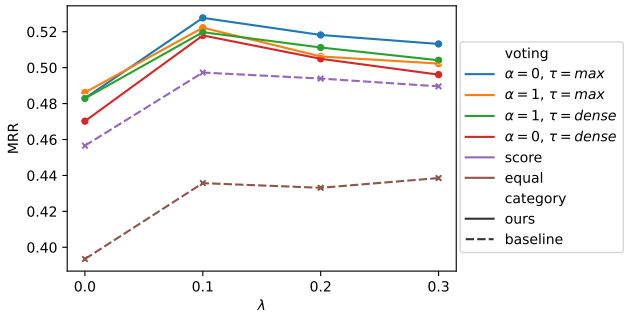

Fig. 6. MRR for each hyperparameter configuration of FONTE

TABLE IV
COMPARISON OF FONTE ($\alpha = 0, \tau = max, \lambda = 0.1$) TO OTHER RANKING TECHNIQUES (# TOTAL SUBJECTS = 130)

	MRR	Accuracy				
		@1	@2	@3	@5	@10
FONTE	**0.528**	**47** (36%)	**66** (51%)	**85** (65%)	**98** (75%)	**110** (85%)
Stage 1&2 of FONTE + Other Techniques (on C_{BIC})						
Bug2Commit	0.380	27	42	64	85	96
FBL-BERT	0.338	27	40	47	69	90
Max Aggr. (Eq. 9)	0.317	17	36	50	73	97
Random	0.218	3	19	41	65	75
Lower Bound	0.145	3	8	19	41	71
Other Techniques (on C)						
Bug2Commit	0.155	11	18	22	25	39
FBL-BERT	0.037	1	3	5	7	10
Ablation Study for FONTE						
Skip Stage 2	0.490	39	64	82	97	110
Use Equal Vote (No FL)	0.436	39	56	67	79	88

hyperparameter setting. The MRRs from our ranking-based voting power (Eq. 4) are plotted in solid lines, while those from the baseline voting schemes, (Eq. 7, and Eq. 8), are plotted in dashed lines. We observe that any hyperparameter setting of FONTE can outperform the baseline voting methods, which demonstrates the effectiveness of using the FL rank of code elements in allocating voting power. Regarding the depth-based decay of voting power, we observe that decay weights $\lambda \in \{0.1, 0.2, 0.3\}$ perform better than $\lambda = 0.0$ (i.e., no decay). In particular, setting λ to 0.1 consistently outperforms other combinations of τ and α.

Table IV shows the comparison between the results of FONTE with its best hyperparameter setting and other baseline ranking techniques in ranking commits in C_{BIC}. We observe that FONTE significantly outperforms all other baselines in terms of all evaluation metrics: it achieves 39% and 56% higher MRR compared to Bug2Commit and FBL-BERT, respectively. The results also show that the performance of Bug2Commit and FBL-BERT can be significantly improved when combined with the search space reduction of FONTE (Stage 1&2). Given the same FL results with FONTE, the max-aggregation method achieves only 60% of MRR compared to FONTE demonstrating the importance of the scoring model.

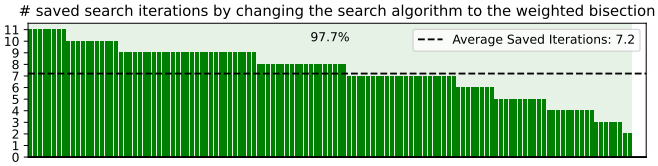

Fig. 7. The number of saved search iterations required until finding the BIC using the weighted bisection compared to the standard bisection on the **entire** commit history, C

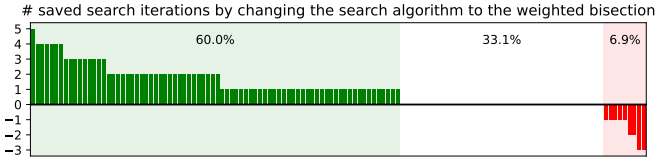

Fig. 8. The number of saved search iterations required until finding the BIC using the weighted bisection compared to the standard bisection on the **reduced** commit history, C_{BIC}

Furthermore, the ablation experiments show that filtering out the style change commits can increase the MRR of FONTE by 8%, showing that Stage 2 is making a significant contribution to the ranking despite the marginal contribution to the search space reduction. However, also note that FONTE still outperforms all other ranking techniques even without Stage 2, which shows the effectiveness of our voting-based scoring model with SBFL. Interestingly, the equal voting model (Eq. 7) with $\lambda = 0.1$, which does not leverage any FL result except the failure coverage, yet performs better than all ranking baselines in terms of MRR. This shows that even without using the FL techniques, simply giving equal voting power to every code element covered by the failing test cases can rank BICs more effectively than the baselines.

Answer to RQ1: FONTE can rank the actual BICs in the top 1 and 5 commits for 36% and 75% of studied bugs. It is significantly more effective than the random baseline and the state-of-the-art IR-based BIC ranking method.

B. *RQ2: Standard Bisection vs. Weighted Bisection*

We simulate the standard and weighted bisection algorithms on all subjects, assuming that the bug-revealing tests can perfectly reveal the existence of bugs. Fig. 7 contains a sorted bar chart that shows the number of saved search iterations, for all subjects, until finding the BIC using our weighted bisection algorithm compared to the standard bisection on the entire commit history. The results show that using the weighted bisection with FONTE-generated scores[10] can reduce the search cost for about 98% of subjects compared to the standard bisection while saving up to 11 search iterations. On average, the number of iterations is reduced by 67%. There is no case where the weighted bisection degrades the performance.

[10]FONTE with $\alpha = 0$, $\tau = max$, $\lambda = 0.1$

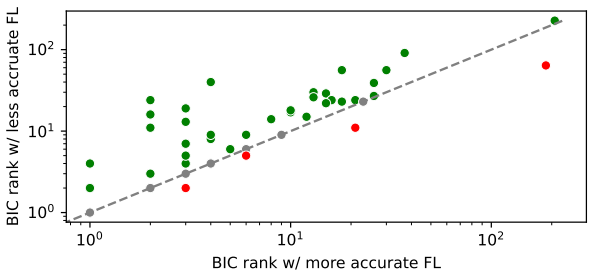

Fig. 9. BIC ranks of FONTE with the more and less accurate FL results

For a more conservative comparison, we also compare the weighted bisection to the standard bisection when both are applied to the reduced search space, C_{BIC}. Fig. 8 shows that the weighted bisection can reduce the number of required search iterations for 78 out of 130 subjects (60.0%), while the number of iterations is increased in only nine out of 130 subjects (6.9%). In the remaining 33.1% of cases, the number of iterations is the same as the standard bisection. To ensure that the median of the number of saved iterations is positive (which would indicate that there is a *performance improvement*), we perform the one-sided Wilcoxon signed rank test [32], whose null hypothesis is that the median of is negative (*performance degradation*). The p-value is $1.51*10^{-11}$, allowing us to reject the null hypothesis in favour of the alternative that *the median of the number of saved iterations is greater than zero*.

We also investigate why the weighted bisection worsens the search efficiency for those nine subjects (6.9%) and report that the BIC is not ranked well in the cases, i.e. either not in the top 10 or even top 50%. The BIC rank (in percentage) and the number of saved iterations are negatively correlated with each other with a Pearson correlation coefficient of -0.58.

Answer to RQ2: Weighted bisection combined with FONTE can save the search cost in 98% of studied bugs compared to the standard bisection applied to the entire commit history, saving 7.2 inspections on average. When the bisection is performed only with the reduced candidates, weighted bisection saves the number of search iterations in 60% of cases while increasing it in only 7% of cases with lower BIC ranks.

C. *RQ3: Impact of FL Accuracy on* FONTE

To see how the accuracy of FL affects the performance of FONTE on each individual subject, we provide less accurate FL results to FONTE and observe how it affects the ranking performance. We intentionally weaken the test suite by removing some of the relevant passing test cases, as it is known that the accuracy of SBFL is highly dependent on the quality of the used test suite [33]. By doing so, we limit the test suite to only the test cases that are contained in the failing test classes. For example, in the case of Table II, the relevant test cases are limited to the last four test cases that are in the `TestFieldUtils` class containing the failing test case.

Among the 99 out of 130 subjects whose sets of relevant test cases are reduced, we observe that, in 57 subjects, the FL accuracy (in terms of the highest rank of buggy methods) is decreased as a result. For those 57 subjects, we see whether the performance of FONTE is affected by the accuracy of FL. In Figure 9, the x- and y-axis represent the BIC ranks produced by FONTE with the more (original) and less accurate FL results, respectively. Green markers (above the dashed line) represent the cases where the better FL yields the better BIC rank, while red markers indicate the opposite. The overall tendency is that higher FL accuracy leads to a better ranking performance of FONTE, as shown by the fact that the number of green dots above the dotted line is much higher than the number of red dots below the line.[11] The one-sided Wilcoxon signed rank test for the paired ranking samples also results in the p-value of $2.56 * 10^{-6}$ showing that the median rank difference is greater than zero when the FL accuracy increases.

Answer to RQ3: FONTE performs better when the FL results used as its input become better. Consequently, we expect that FONTE can benefit from more precise and sophisticated FL techniques in the future.

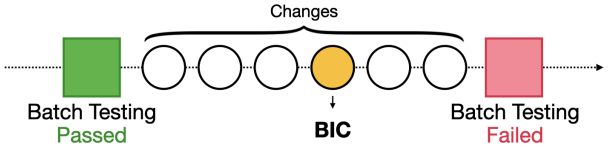

Fig. 10. Simplified batch testing scenario

VII. APPLICATION TO INDUSTRY SOFTWARE

SAP HANA[12] is a large-scale commercial software that consists of more than 10M lines of C++ and C. In the CI system of SAP HANA, multiple commits that have individually passed the pre-submit testing are merged into the delivery branch and tested together using a more extensive test suite on a daily basis. Considering the set of multiple commits as a single batch, this is a type of *Batch Testing* [34]. While batch testing reduces the overall test execution cost for SAP HANA, it also has some practical drawbacks: when a test fails, it is not immediately clear which change in the batch is responsible for the failure [35]. The current CI system of SAP HANA identifies the BIC in the batch using automatic bisection to aid the bug assignments. However, each individual inspection during the bisection can take up to several hours due to the compilation, installation, and test execution cost, resulting in severe bottlenecks in the overall debugging process. The bottleneck can be particularly problematic if integration or system-level tests fail.

This motivates us to see whether FONTE and its weighted bisection can reduce the number of bisection iterations. To evaluate the effectiveness of applying FONTE, we collect 23 batch testing failures that occurred from July to August 2022 and their BICs identified by the bisection from the internal CI logs of SAP HANA. Using the data, we first

[11] FONTE with the worse FL results still outperforms all baselines in RQ1.
[12] The product name is anonymised due to DBR.

TABLE V
EVALUATION OF FONTE ON THE 23 BATCH TESTING FAILURES OF SAP HANA

	MRR	Accuracy				
		@1	@2	@3	@5	@10
FONTE	0.600	10 (43%)	14 (61%)	15 (65%)	20 (87%)	23 (100%)
Random	0.110	0	0	0	1	17

check if FONTE can find the BIC inside the batch accurately (Fig. 10). As the test coverage of SAP HANA is regularly and separately updated, instead of being measured at each of the batch testings, we use the latest line-level coverage information to calculate the Ochiai scores. Note that we do not need to compute the Ochiai scores for all lines, but only the lines covered by the failing tests. When applying FONTE, depth-based voting decay is not used ($\lambda = 0$) because all candidate commits are submitted on the same day and have not yet been merged into the main codebase. For the remaining hyperparameters, we use $\alpha = 1$ and $\tau = max$ that performed the best with $\lambda = 0$ in RQ1.

Table V shows the BIC ranking performance of FONTE in terms of MRR and Accuracy@n. While each batch contains 18.48 commits on average, FONTE can locate the actual BIC within the top 1 and 5 for 43% and 87% of the failures, respectively. Compared to the random baseline, it achieves 5.5-fold increase in MRR. Further, we also report that the weighted bisection can reduce the bisection iterations for 18 out of 23 cases (78%), while it increases the cost in only three cases (13%). Based on this result, we plan to incorporate weighted bisection into the CI process of SAP HANA, which is expected to save 32% of required iterations. Considering that each iteration can take up to several hours, we expect a significant reduction in the average BIC identification cost for SAP HANA in the long run.

VIII. THREATS TO VALIDITY

Threats to internal validity concern factors that can affect how confident we are about the causal relationship between the treated factors and the effects. FONTE relies on widely-adopted open-source tools to establish Cover and Evolve relations to ensure the chain of causality between the test failure and BIC identification. Other baselines depend on multiple sources of information, such as bug reports. We choose Defects4J as our benchmark as it provides well-established links between real-world bug reports and the buggy version, not to mention human-written bug-revealing test cases that withheld scrutiny from the community.

Threats to external validity concern factors that may affect how well our findings can be generalised to unseen cases. Our key findings are primarily based on experiments with the open-source Java programs in Defects4J. Since they are not representative of the entire population of real-world programs, only further evaluations can strengthen our claim of generalisation. We tried to support our claim by evaluating FONTE with industry-scale software written in C and C++. We do note that FONTE does not generalise to bugs that are caused by non-executable files, such as configuration changes, as its base assumption is that the test failure is caused by a bug in the source code. We leave extension of FONTE to bugs caused by non-executable changes as our primary future work.

Threats to construct validity concern how well the used metrics measure the properties we aim to evaluate. We adopt two ranking-evaluation metrics, MRR and accuracy@n, to evaluate FONTE: both have been widely used in the IR and SE literature. Since they are based on absolute ranks, we do note that the results can be overrated when the number of ranking candidates is small. To mitigate the threat, we also present the expected and worst values for the measures as baselines.

IX. RELATED WORK

Locus [5] is the first work that proposed to localise the bug at the software change level. It takes a bug report as an input query and locates the relevant change hunk based on the token similarities. IR-based techniques, such as Locus, and FONTE can complement each other depending on circumstances. When the failure cannot be reproduced from the bug report, IR-based techniques can be used instead of FONTE. However, if the coverage of the failing and passing tests are available, we can apply FONTE with SBFL to more precisely rank the commits without relying on IR.

ChangeLocator [2] aims to find a BIC for crashes using the call stack information. It is a learning-based approach that requires data from fixed crashes. Unlike ChangeLocator, FONTE is not limited to crashes and can be applied to general failures. Orca [10] takes symptoms of bugs, such as an exception message or customer complaints, as an input query and outputs a ranked list of commits ordered by their relevance to the query. It uses the TF-IQF [36] to compute the relevance scores of files, and aggregate them to a commit level. Subsequently, it uses machine learning to predict the risk of candidate commits for breaking ties. Bug2Commit [6] uses multiple features extracted from bug reports and commits, and aggregates all features by taking the average of their vector representations. Although Bug2Commit uses an unsupervised learning approach, it needs the historical data of project-specific bug reports and commits to train the word embedding model. FBL-BERT [28] retrieves the relevant changeset for the input bug report using a fine-tuned BERT model that can capture the semantics in the text. It proposes fine-grained changeset encoding methods and accelerates the retrieval by offline indexing [37]. The major difference between FONTE and the techniques mentioned above is that FONTE does not require any training. Further, FONTE can be combined with any code-level FL technique, without being coupled to specific sources of information, as long as the coverage of failing executions is available.

FaultLocator [38] is similar to FONTE as both use code-level FL scores to identify suspicious changes. FaultLocator combines spectrum information with the change impact analysis to precisely identify the failure-inducing *atomic* edits out

of all edits between two versions, whereas FONTE aims to pinpoint BICs in the commit history.

X. CONCLUSION

This paper proposes FONTE, a BIC identification technique that is available upon the observation of a failure. It prunes the BIC search space using failure coverage and the syntactic analysis of commits, and assigns scores to the remaining commits using the FL scores as well as change histories of code elements. Our experiments with 130 bugs in Defects4J show that FONTE can effectively identify BICs with an MRR of 0.528, which significantly outperforms the baselines including state-of-the-art BIC identification techniques. Along with FONTE, we also propose the weighted bisection to accelerate the BIC search utilising the commit score information and show that it can save the search cost in 97.7% of the studied cases compared to the standard bisection. Finally, the application of FONTE to a large-scale industry software SAP HANA shows that FONTE can successfully reduce the cost of BIC identification in a batch testing CI scenario. Future work includes the actual deployment of FONTE to SAP HANA as well as expanding the scope of bugs FONTE can handle.

ACKNOWLEDGMENT

Gabin An and Shin Yoo are supported by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629), Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001), and Samsung Electronics (Grant No. IO201210-07969-01).

REFERENCES

[1] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, jul 2005. [Online]. Available: https://doi.org/10.1145%2F1082983.1083147

[2] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang, "ChangeLocator: locate crash-inducing changes based on crash reports," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2866–2900, nov 2017. [Online]. Available: https://doi.org/10.1007%2Fs10664-017-9567-4

[3] M. Wen, Y. Liu, and S.-C. Cheung, "Boosting automated program repair with bug-inducing commits," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, jun 2020. [Online]. Available: https://doi.org/10.1145%2F3377816.3381743

[4] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proceedings of the 27th ACM European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, aug 2019. [Online]. Available: https://doi.org/10.1145%2F3338906.3338962

[5] M. Wen, R. Wu, and S.-C. Cheung, "Locus: locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, aug 2016. [Online]. Available: https://doi.org/10.1145%2F2970276.2970359

[6] V. Murali, L. Gross, R. Qian, and S. Chandra, "Industry-scale IR-based bug localization: A perspective from facebook," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, may 2021. [Online]. Available: https://doi.org/10.1109%2Ficse-seip52600.2021.00028

[7] 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 Software Engineering*, vol. 25, no. 2, pp. 1294–1340, feb 2020. [Online]. Available: https://doi.org/10.1007%2Fs10664-019-09781-y

[8] C. Catal, "Software fault prediction: A literature review and current trends," *Expert Systems with Applications*, vol. 38, no. 4, pp. 4626–4636, apr 2011. [Online]. Available: https://doi.org/10.1016%2Fj.eswa.2010.10.024

[9] "Git-bisect-lk2009 documentation." [Online]. Available: https://git-scm.com/docs/git-bisect-lk2009.html

[10] R. Bhagwan, R. Kumar, C. S. Maddila, and A. A. Philip, "Orca: Differential bug localization in large-scale services," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 493–509.

[11] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, aug 2016. [Online]. Available: https://doi.org/10.1109%2Ftse.2016.2521368

[12] G. An and S. Yoo, "Reducing the search space of bug inducing commits using failure coverage," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, aug 2021. [Online]. Available: https://doi.org/10.1145%2F3468264.3473129

[13] S. Artzi, S. Kim, and M. D. Ernst, "ReCrash: Making software failures reproducible by preserving object states," in *ECOOP 2008 – Object-Oriented Programming*. Springer Berlin Heidelberg, pp. 542–565. [Online]. Available: https://doi.org/10.1007%2F978-3-540-70592-5_23

[14] W. Jin and A. Orso, "BugRedux: Reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012. [Online]. Available: https://doi.org/10.1109%2Ficse.2012.6227168

[15] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, sep 2010. [Online]. Available: https://doi.org/10.1109%2Ftse.2010.63

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, 2014. [Online]. Available: https://doi.org/10.1145%2F2610384.2628055

[17] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, sep 2006. [Online]. Available: https://doi.org/10.1109%2Fase.2006.23

[18] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[19] J. Sohn and S. Yoo, "Why train-and-select when you can use them all?" in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, jul 2019. [Online]. Available: https://doi.org/10.1145%2F3321707.3321873

[20] J. Sohn, G. An, J. Hong, D. Hwang, and S. Yoo, "Assisting bug report assignment using automated fault localisation: An industrial case study," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2021. [Online]. Available: https://doi.org/10.1109%2Ficst49551.2021.00041

[21] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. Le Traon, "What made this test flake? pinpointing classes responsible for test flakiness," *arXiv e-prints*, pp. arXiv–2207, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2207.10143

[22] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, "CodeShovel: Constructing method-level source code histories," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, may 2021. [Online]. Available: https://doi.org/10.1109%2Ficse43902.2021.00135

[23] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*. ACM Press, 2007. [Online]. Available: https://doi.org/10.1145%2F1250734.1250782

[24] R. Abreu, P. Zoeteweij, and A. V. Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006. [Online]. Available: https://doi.org/10.1109%2Fprdc.2006.18

[25] N. Craswell, "Mean reciprocal rank," in *Encyclopedia of Database Systems*. Springer US, 2009, pp. 1703–1703. [Online]. Available: https://doi.org/10.1007%2F978-0-387-39940-9_488

[26] J. Sohn and S. Yoo, "FLUCCS: using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2017. [Online]. Available: https://doi.org/10.1145%2F3092703.3092717

[27] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2020. [Online]. Available: https://doi.org/10.1145%2F3395363.3397351

[28] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with BERT," in *Proceedings of the 44th International Conference on Software Engineering*. ACM, may 2022. [Online]. Available: https://doi.org/10.1145%2F3510003.3510042

[29] J. Tabassum, M. Maddela, W. Xu, and A. Ritter, "Code and named entity recognition in StackOverflow," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4913–4926. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.443

[30] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford *et al.*, "Okapi at TREC-3," *Nist Special Publication Sp*, vol. 109, p. 109, 1995.

[31] M. Hucka, "Spiral: splitters for identifiers in source code files," *Journal of Open Source Software*, vol. 3, no. 24, p. 653, Apr. 2018. [Online]. Available: http://joss.theoj.org/papers/10.21105/joss.00653

[32] F. Wilcoxon, "Individual comparisons by ranking methods," in *Springer Series in Statistics*. Springer New York, 1992, pp. 196–202. [Online]. Available: https://doi.org/10.1007%2F978-1-4612-4380-9_16

[33] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, may 2017. [Online]. Available: https://doi.org/10.1109%2Ficse.2017.66

[34] A. Najafi, P. C. Rigby, and W. Shang, "Bisecting commits and modeling commit risk during testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, aug 2019. [Online]. Available: https://doi.org/10.1145%2F3338906.3338944

[35] M. J. Beheshtian, A. H. Bavand, and P. C. Rigby, "Software batch testing to save build test resources and to reduce feedback time," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2784–2801, aug 2022. [Online]. Available: https://doi.org/10.1109%2Ftse.2021.3070269

[36] J.-M. Yang, R. Cai, F. Jing, S. Wang, L. Zhang, and W.-Y. Ma, "Search-based query suggestion," in *Proceeding of the 17th ACM conference on Information and knowledge mining - CIKM '08*. ACM Press, 2008. [Online]. Available: https://doi.org/10.1145%2F1458082.1458321

[37] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019. [Online]. Available: https://doi.org/10.1109/TBDATA.2019.2921572

[38] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 23–32. [Online]. Available: https://doi.org/10.1109/ICSM.2011.6080769