Reducing the Search Space of Bug Inducing Commits using Failure Coverage

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

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

ABSTRACT

Knowing how exactly a bug has been introduced into the code can help developers debug the bug efficiently. However, techniques currently used to retrieve Bug Inducing Commits (BICs) from the repository timeline are limited in their accuracy. Automated bisection of the version history depends on the bug revealing test case being executable against all candidate previous versions, whereas blaming the last commits that touched the same parts as the fixing commit (à la SZZ) requires that the bug has already been fixed. We show that filtering commits using the coverage of the bug revealing test cases can effectively reduce the search space for both bisection and SZZ-like blame models by 87.6% and 27.9%, respectively, significantly reducing the cost of BIC retrieval. The application of our approach to bugs in Defects4J also reveals inconsistencies in some of their BICs known in the literature.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**.

KEYWORDS

bug inducing commit, test coverage

ACM Reference Format:
Gabin An and Shin Yoo. 2021. Reducing the Search Space of Bug Inducing Commits using Failure Coverage. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3468264.3473129

1 INTRODUCTION

Given a bug in an evolving software system, a Bug Inducing Commit (BIC) refers to the specific commit that introduced the bug into the system [8]. Identifying and collecting BICs can be beneficial in many ways [2]. An existing survey of Microsoft developers reveals that over 75% of those who responded have used version history when debugging their code [9]. In particular, over 90% of those who said version history helps specifically pointed out BICs as the most useful information from version histories. It has also been suggested that the knowledge of BICs can boost automated debugging [10, 12].

Unfortunately, there is no technique that can identify BICs from the version history for debugging activities both accurately and efficiently. The standard approach for finding a specific commit that introduced a bug is *bisecting*, i.e., to perform a binary search between the known *good* and *bad* commits, inspecting whether each successive middle commit is buggy or not. Despite the support from version control systems (e.g., `git bisect`[1]), this is often expensive due to a couple of reasons. First, the inspection may have to be manual, as the bug revealing tests may not always be executable against the previous versions. Second, if the last good commit is not explicitly known, one may have to perform bisection over the complete version history, resulting in numerous inspections.

A closely related BIC identification technique is the popular SZZ [8] and its variants, which require a Bug Fixing Commit (BFC) as the starting point. Intuitively, given a BFC, SZZ returns a set of commits that last modified each element of the BFC, under the assumption that BFC aims to fix a problem in those commits. Compared to bisecting, this establishes only traceability and not the bugginess of the retrieved commits (the corresponding feature of `git` would be `git blame`[2]). Many *filters* have been proposed to identify and exclude intermediate commits that have little to do with the actual bug: for example, RA-SZZ skips any refactoring commits [6]. The lack of explicit inspection, as well as the heuristic nature of filtering, can result in low precision in BIC identification [3, 4, 12]. Furthermore, SZZ and its variants are not applicable in a debugging scenario, because the fix does not exist yet. IR based techniques share a similar limitation due to the lack of bug reports at debugging time [11, 14].

We propose a technique that can reduce the bisection search space for BIC dramatically. The technique is based on our observation that BICs are likely to be traceable from the parts of the program covered by the bug revealing executions. Consequently, the bisection only needs to consider the commits that are traceable from the coverage of bug revealing tests. We show that we can easily filter out commits that are unrelated to a given failing execution. An empirical evaluation on Defects4J shows that the search space is reduced to only 12.4% of the complete version history from the beginning up to the buggy snapshot, not only reducing the cost of manual bisection but also improving the precision of SZZ.

2 BIC SEARCH SPACE REDUCTION

2.1 A Motivating Example

Listing 1 contains commit `c95eb0b` (nicknamed `Cactus`) made to the Apache Commons Math project: `Cactus` has been identified as

[1]https://git-scm.com/docs/git-bisect
[2]https://git-scm.com/docs/git-blame

```
1 --- a/src/java/org/apache/commons/math/ode/nonstiff/
      AdaptiveStepsizeIntegrator.java
2 +++ b/src/java/org/apache/commons/math/ode/nonstiff/
      AdaptiveStepsizeIntegrator.java
3 @@ -108,8 +108,8 @@ public abstract class
      AdaptiveStepsizeIntegrator
4      this.scalAbsoluteTolerance = 0;
5      this.scalRelativeTolerance = 0;
6 -    this.vecAbsoluteTolerance  = vecAbsoluteTolerance;
7 -    this.vecRelativeTolerance  = vecRelativeTolerance;
8 +    this.vecAbsoluteTolerance  = vecAbsoluteTolerance.clone();
9 +    this.vecRelativeTolerance  = vecRelativeTolerance.clone();
10
11     resetInternalState();
```

Listing 1: Changes introduced by `c95eb0b` that is identified as a BIC of Math-74 by Wen et al. [12]

```
<method name="&lt;init&gt;" signature="(Ljava/lang/String;DD[D[D
    )V" line-rate="0.0" branch-rate="1.0">
    <lines>
        <line number="123" hits="0" branch="false"/>
        // ... skipped ...
        <line number="131" hits="0" branch="false"/> //*
        <line number="132" hits="0" branch="false"/> //*
        <line number="134" hits="0" branch="false"/>
        <line number="136" hits="0" branch="false"/>
    </lines>
</method>
```

Listing 2: Coverage results of the bug-revealing test of Math-74, `AdamsMoultonIntegratorTest:polynomial`, on the initialiser of the class `AdaptiveStepsizeIntegrator`.

the BIC for the bug `Math-74` in Defects4J, as part of a BIC dataset for 91 bugs in Defects4J recently released by Wen et al. [12].[3] `Cactus` changes the class named `AdaptiveStepsizeIntegrator` by appending `".clone()"` to two of its statements.

Defects4J provides a single bug revealing test case for `Math-74`. Part of the coverage measured from its failure against the buggy snapshot of `Math-74`, `034b4d6` (nicknamed `Burger`), is shown in Listing 2. The two `line` elements marked with `*` correspond to the two lines added by `Cactus`, respectively (Line 8-9 in Listing 1), and the attribute `hits` stores the number of times the corresponding element has been executed. From this, we find that the lines added by `Cactus` are not executed by the failing test case on the buggy snapshot, which suggests that the change in `Cactus` may not be the cause of the failure in `Math-74`. This can be verified by the fact that the failure still occurs when `Cactus` is reverted on `Burger`.

From this observation of inconsistency, we posit a simple yet effective and important necessary condition for a BIC: ***a bug-inducing commit should change what eventually is covered by failing executions***. Using this condition, we can eliminate commits that are irrelevant to the specific bug being analysed.

2.2 Efficient BIC Search using Failure Coverage

Our motivating example shows that the *failure coverage*, the code coverage of failing executions, can be used to filter out the commits that are irrelevant to the currently observed failure. This filtering approach can reduce the search space for both bisection and the SZZ algorithm and is available at debugging time, because it only requires the execution traces of the failing test cases and not the availability of the fixing commit like SZZ.

[3]https://github.com/justinwm/InduceBenchmark/blob/master/Defects4J.csv

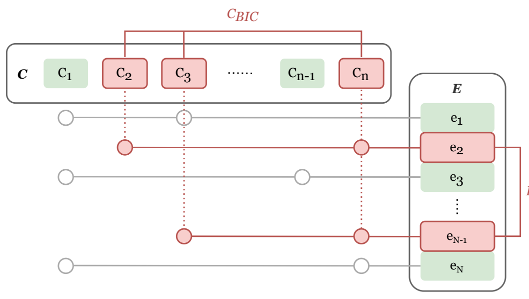


Figure 1: Reducing the BIC candidates using failure coverage

Consider a BIC c for a bug b: the changes introduced by c should eventually result in the program failure caused by b. Except for omission faults, i.e., the non-existence of needed code, a failing execution should cover at least one of the faulty program elements (note that this is the same assumption shared by fault localisation techniques). If we are to suspect all program elements covered by failing test cases as potential candidates of the fault, it follows that commits not involved in the evolution of those elements can be safely filtered out.

More formally, let $C = \{C_1, \ldots, C_n\}$ be a set of commits for a program P such that C_i is older than C_{i+1}. Suppose $E = \{e_1, \ldots, e_N\}$ is the set of executable program elements of P, and $T_F = \{t_1, \ldots, t_M\}$ is the set of *failing* test cases for P. Here, we assume that P is a faulty program and T_F is non-empty. For each failing test case $t \in T_F$, let $E_t \subseteq E$ denote the set of program elements covered by t. If the fault that caused the failure is not an omission fault, for all $t \in T_F$, at least one of the program elements in E_t should be faulty. Since we do not exactly know which of them are faulty, we may suspect the entire E_t as fault candidates. Therefore, all possible fault candidates, E_F, can be defined by $E_F = \bigcup_{t \in T_F} E_t$.

In a version control system such as `Git`, one can trace the set of commits that are involved in the evolution of a specific source code line. By leveraging that functionality, given a program element e, we define $C_e \subseteq C$ as the set of commits that either created or modified e. For the fault candidates E_F, we regard all commits involved in the evolution of at least one of the faulty candidates as bug-inducing candidates. Therefore, our reduced search space of BIC can be defined as $C_{BIC} = \bigcup_{e \in E_F} C_e$, as depicted in Figure 1. In terms of identifying non-BIC, this analysis is theoretically sound: if a commit c is not in C_{BIC}, then c is not a BIC.

3 EVALUATION ON DEFECTS4J

We apply our BIC search space reduction method to Defects4J [5] version 2.0.0, a real-world Java faults dataset. It contains a total of 835 bugs from various open-source projects, with a set of bug-revealing test cases for each bug. Table 1 shows the number of faults included for each of the 16 projects, as well as the average number of commits that precede the faulty snapshot. We have excluded `Chart` because its version control system is not `Git`. Note that we apply our method on the *real* buggy version,[4] and not the buggy version in Defects4J that only contains the isolated and minimised

[4]One can get `"revision.id.buggy"` (commit hash of the buggy version) using `"defects4j query"` command.

Table 1: Application subjects in DEFECTS4J. (*JS*: *Jackson*)

Project	#Faults Used / All	Avg. #Commits	Project	#Faults Used / All	Avg. #Commits
Cli	32 / 39	493.7	Closure	173 / 174	1421.7
Codec	18 / 18	922.2	Collections	4 / 4	2816.0
Compress	47 / 47	1387.8	Csv	16 / 16	821.6
Gson	18 / 18	1218.4	*JS*Core	25 / 26	1218.4
*JS*Databind	111 / 112	3025.8	*JS*Xml	6 / 6	656.3
Jsoup	8 / 93	1195.2	JxPath	22 / 22	368.1
Lang	64 / 64	2393.3	Math	106 / 106	2928.7
Mockito	33 / 38	1828.0	Time	20 / 26	1624.4

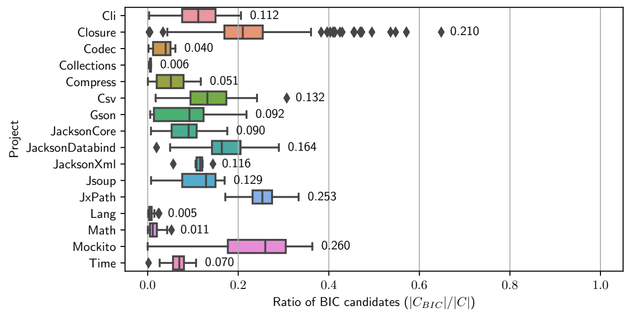


Figure 2: Ratio of BIC candidates for each project. The text on the right side of each box represents the median value.

buggy change. Therefore, any buggy version against which we cannot execute the bug revealing tests provided by DEFECTS4J has also been excluded. In total, we use 703 faults in our evaluation. Our replication package, which is publicly available[5], contains the reason for excluding each subject. The remainder of this section contains the implementation details and the experimental findings.

3.1 Implementation Details

`Git` can produce the entire list of commits that have modified code lines with line numbers in the range [*begin*, *end*] with the following command: `git log -C -M -L [begin],[end]:[path_to_file]`. The options `-C` and `-M` allow us to track the copies and renames of files, respectively. Let us consider our motivating example. By running this command on the buggy snapshot of `Math-74` (`Burger`), we can retrieve two commits that modified the lines 131 and 132 corresponding to Line 5-6 in Listing 2. The first commit (`89ac173`) introduced the original code lines (Line 6-7 in Listing 1), while the second one (`Cactus`) changed the lines (Line 8-9 in Listing 1). It is worth noting that the option `-M` enables us to retrieve the commits even though the file path at the commit time was different from the one in `Burger`. Furthermore, we implement our approach at a ***method***-level granularity. To do so, we first measure the statement-level coverage using Cobertura and subsequently use JavaParser to extract the line range of the method surrounding each statement. [6]

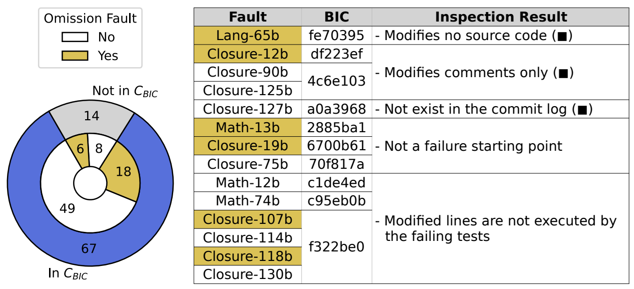
3.2 Result and Discussion

This section presents the results of failure-coverage-based filtering to BIC retrieval for DEFECTS4J faults.

3.2.1 Search Space Reduction. Boxplots in Figure 2 show the ratio of BIC search space reduction, i.e., $|C_{BIC}|/|C|$, for bugs in each project. On average, using the failure coverage, we can filter out 87.6% of commits from the search space. In case of `Collections` and `Lang`, the BIC candidates are only 0.6% and 0.5% of the total commits up to the buggy version, respectively. For example, while the buggy version of `Lang-51` has 1,682 preceding commits, there are only two commits in C_{BIC}, showing that the failure coverage can drastically narrow down the BIC candidates.

Finding 1: Leveraging only the coverage of failing test cases, we can dramatically reduce the BIC search space by 87.6% on average. The reduction can ultimately improve the efficiency of any BIC retrieval techniques, such as bisection or SZZ.

[5] **https://github.com/coinse/FSE2021-IVR-BIC**
[6] **Cobertura**: https://cobertura.github.io/cobertura/, **JavaParser**: http://javaparser.org



Fault	BIC	Inspection Result
🟨 Lang-65b	fe70395	- Modifies no source code (■)
🟨 Closure-12b	df223ef	- Modifies comments only (■)
Closure-90b	4c6e103	
Closure-125b		
Closure-127b	a0a3968	- Not exist in the commit log (■)
🟨 Math-13b	2885ba1	- Not a failure starting point
🟨 Closure-19b	6700b61	
Closure-75b	70f817a	
Math-12b	c1de4ed	- Modified lines are not executed by the failing tests
Math-74b	c95eb0b	
🟨 Closure-107b	f322be0	
Closure-114b		
🟨 Closure-118b		
Closure-130b		

Figure 3: Inspection results for the commits that were identified as BIC [12] but not contained in C_{BIC}

3.2.2 Soundness Verification. To verify the soundness of our filtering technique, we use the existing curated BIC dataset provided by Wen et al. [12] (referred to as *Wen-BIC*). Among the 91 bugs in the dataset, we have excluded nine from `Chart` and one from `Time`, that are not in our application subjects. For the remaining 81 bugs, we check whether our reduced BIC search space, C_{BIC}, contains the Wen-BIC or not.

The pie chart in Figure 3 shows that 67 out of 81 Wen-BICs are found in C_{BIC} (marked in blue), whereas the remaining 14 Wen-BICs are filtered out by our technique. We manually inspected the 14 cases to evaluate the soundness of the filtering and conclude that all BICs in those 14 cases are not real BICs. The table in Figure 3 presents the inspection results. For five Wen-BICs (marked with ■), based on the lines they changed, we can confirm that the commits have not, in fact, introduced the bug. For example, `4c6e103`, which was identified as the BIC of bugs `90` and `125` of `Closure`, only modifies *comments* of multiple files without changing any source code. Since it does not introduce any semantic changes, it cannot be a BIC. In another case, the commit `a0a3968`, which was identified as the BIC of `Closure-127`, does not exist in the commit history of the project. The remaining nine commits present more challenging cases that are difficult to conclude based on the inspection of the changed lines only. However, the empirical evidence still suggests that these are not BICs, either because the bug revealing test cases fail on preceding snapshots too, or because they modify lines that are not covered by the bug revealing test cases. Detailed results of the manual inspection are available in our public repository.

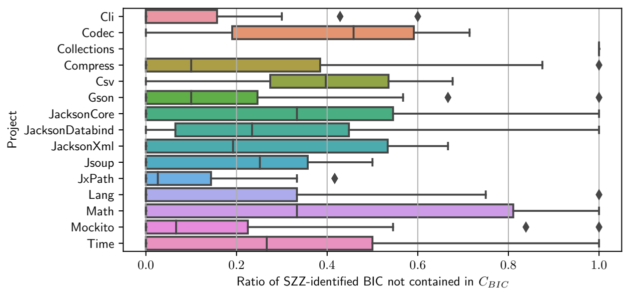


Figure 4: Ratio of SZZ-identified BIC that are not contained in our BIC candidates

Finding 2: Our investigation of the existing BIC ground-truth dataset shows that we can safely filter out irrelevant commits from the search space for BICs. The results also reveal that some of the ground-truth BICs in the dataset cannot be BICs.

3.2.3 Improving SZZ. SZZ and its variants can be imprecise [1, 3, 7, 12]. We investigate whether our filtering technique can improve the precision of the SZZ algorithm. We apply `SZZUnleashed` [2], an open-source implementation of the SZZ [8] algorithm with *line number mappings* [13], to the commits that correspond to the fixed versions identified by Defects4J. Using the associated bug reports to extract information required by `SZZUnleashed`, such as report creation date or bug resolution date, we successfully collected data for 524 out of 703 bugs in Table 1. Let C_{SZZ} denote the set of BICs retrieved by `SZZUnleashed`. Figure 4 shows the ratio of commits in C_{SZZ} which are not in C_{BIC}, i.e., $\frac{|C_{SZZ} \setminus C_{BIC}|}{|C_{SZZ}|}$. On average, 27.9% of commits that have been blamed by `SZZUnleashed` are filtered out by our technique, confirming that the precision of `SZZUnleashed` is less than 72.1% on average. Combined with Finding 2, this shows that utilising failure coverage can safely eliminate the irrelevant commits found by the other BIC retrieval technique.

Finding 3: Our dynamic analysis not only increases the efficiency of BIC search in a debugging phase but can also improve the precision of the SZZ algorithm when the BFC is known.

3.2.4 Discussion of Omission Faults and Their BICs. Figure 3 shows that BICs of omission faults are found in C_{BIC}. Since omission faults refer to the absence of needed code, the concept of their BICs, as well as why they are in C_{BIC}, requires some discussion. The nature of omission faults implies that certain code has been omitted, while other closely related code lines being added or modified. Consequently, we can define the BIC of an omission fault as the commit in which the related lines were added. Failing test cases are likely to execute those closely related lines, but not the omitted lines, hence the failure. We conjecture that this is how and why we can include the BICs of omission faults in C_{BIC}.

Consider `Lang-51` as an example: it is an omission fault that can be fixed by adding one line to the method `toBoolean` in the class `BooleanUtils`. Since both neighbouring lines of the omitted line are executed by failing test cases, the ground-truth BIC `49b8c60` that was introduced the lines can be found in C_{BIC}.

4 CONCLUSION AND FUTURE WORK

We propose using the coverage of failing tests to filter out commits irrelevant to the program elements that may have cause the test failures. An evaluation using Defects4J shows that our filtering technique can significantly reduce the search space of BICs, with over 87% of all commits being safely excluded from the search for BICs. Our filtering also shows that some BICs reported in the literature may be incorrect. While the proposed technique may not be capable of pinpointing a single BIC, the preliminary evaluation shows that dynamic information can improve BIC retrieval. For future work, we will consider additional dynamic information, such as fault localisation results, to further refine and rank BIC candidates.

ACKNOWLEDGMENTS

Authors thank Marcel Böhme for the insightful discussions about BICs. Gabin An and Shin Yoo have been supported by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629), as well as by Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001).

REFERENCES

[1] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In *Proceedings of International Symposium on Software Testing and Analysis*. 105–115.
[2] 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*. 7–12.
[3] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.
[4] Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. 2019. Issues with SZZ: An empirical assessment of the state of practice of defect prediction data collection. *arXiv preprint arXiv:1911.08938* (2019).
[5] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.
[6] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. 380–390.
[7] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. arXiv:2102.03300 [cs.SE]
[8] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.
[9] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* (2019).
[10] Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2020. Boosting automated program repair with bug-inducing commits. In *Proceedings of the 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 77–80.
[11] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 262–273.
[12] 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 326–337.
[13] Chadd Williams and Jaime Spacco. 2008. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*. 32–36.
[14] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23, 5 (2018), 2866–2900.