iBiR: Bug-report-driven Fault Injection

AHMED KHANFIR, SnT, University of Luxembourg, Luxembourg
ANIL KOYUNCU, Sabanci University, Turkey
MIKE PAPADAKIS, MAXIME CORDY, TEGAWENDE F. BISSYANDÉ, JACQUES KLEIN, and YVES LE TRAON, SnT, University of Luxembourg, Luxembourg

Much research on software engineering relies on experimental studies based on fault injection. Fault injection, however, is not often relevant to emulate real-world software faults since it "blindly" injects large numbers of faults. It remains indeed challenging to inject few but realistic faults that target a particular functionality in a program. In this work, we introduce iBiR, a fault injection tool that addresses this challenge by exploring change patterns associated to user-reported faults. To inject realistic faults, we create mutants by re-targeting a bug-report-driven automated program repair system, i.e., reversing its code transformation templates. iBiR is further appealing in practice since it requires deep knowledge of neither code nor tests, just of the program's relevant bug reports. Thus, our approach focuses the fault injection on the feature targeted by the bug report. We assess iBiR by considering the Defects4J dataset. Experimental results show that our approach outperforms the fault injection performed by traditional mutation testing in terms of semantic similarity with the original bug, when applied at either system or class levels of granularity, and provides better, statistically significant estimations of test effectiveness (fault detection). Additionally, when injecting 100 faults, iBiR injects faults that couple with the real ones in around 36% of the cases, while mutation testing achieves less than 4%.

CCS Concepts: • **Software and its engineering → Software fault tolerance**; *Software reliability*; Software usability; Software performance; *Software safety*; *Correctness*; **Software maintenance tools**; *Software testing and debugging*; **Software verification and validation**; **Software defect analysis**; **Maintaining software**; **Software evolution**; Software version control; **Software post-development issues;**

Additional Key Words and Phrases: Fault injection, mutation, bug reports, information retrieval

ACM Reference format:
Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon. 2023. iBiR: Bug-report-driven Fault Injection. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 33 (March 2023), 31 pages.
https://doi.org/10.1145/3542946

1 INTRODUCTION

A key challenge of fault injection techniques (such as mutation analysis) is to emulate the effects of real faults. This property of representativeness of the injected faults is of particular importance

This work was supported by the Luxembourg National Research Fund (FNR) TestFast Project, ref. 12630949 and partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949014 for project NATURAL).
Authors' addresses: A. Khanfir, M. Papadakis, M. Cordy, T. F. Bissyandé, J. Klein, and Y. Le Traon, SnT, University of Luxembourg, 29 Avenue John F. Kennedy, Luxembourg, Luxembourg, 1855; emails: {ahmed.khanfir, michail.papadakis, maxime.cordy, tegawende.bissyande, jacques.klein, Yves.LeTraon}@uni.lu; A. Koyuncu, Sabanci University, Ãœniversite Caddesi No: 27, Istanbul, Turkey, 34956; email: anil.koyuncu@sabanciuniv.edu.

1049-331X/2023/03-ART33 $15.00
https://doi.org/10.1145/3542946

since fault injection techniques are widely used by researchers when evaluating and comparing bug finding, testing, and debugging techniques, e.g., test generation, bug fixing, fault localization, and so forth [60]. This means that there is a high risk of mistakenly asserting test effectiveness in case the injected faults are non-representative.

Typically, fault injection techniques introduce faults by making syntactic changes in the target programs' code using a set of simple syntactic transformations [14, 33, 51], usually called mutation operators. These transformations have been defined based on the language syntax [4] and are "blindly" mutating the entire codebase of the projects, injecting large numbers of mutants, with the hope to inject some realistic faults. This means that there is a limited control on the fault types and the locations where to inject faults. In other words, the appropriate "what" and "where" to inject faults in order to make representative fault injection have been largely ignored by existing research.

Fault injection techniques may also draw on recent research that mines fault patterns [8, 70] and demonstrate some form of realism w.r.t. real faults. These results indicate that the injected faults may carry over the realism of the patterns, a fact that removes a potential validity threat. However, at the same time, they are limited as they do not provide any control on the locations and target functionality, thus impacting fault representativeness [9, 51, 62].

This is an important limitation especially for large real-world systems for the following two reasons: (1) injecting faults everywhere escalates the application cost due to the large number of mutants introduced and (2) the results could be misleading since a tiny ratio of the injected faults are coupled to the real ones [62] and the injected set of faults does not represent the likelihood of faults appearing in the field [51]. Therefore, representativeness of the injected faults in terms of fault types and locations is of utmost importance w.r.t. both application cost and accuracy of the method.

To bypass these issues, one could use real faults (mined from the projects' repositories) or directly apply the testing approach to a set of programs and manually identify potential faults. While such a solution brings realism into the evaluations, it is often limited to a few fault instances (of limited diversity), requires an expensive manual effort in identifying the faults, and fails to offer the experimental control required by many evaluation scenarios.

We advance in this research direction by bringing realism in the fault injection via leveraging information from bug reports. Bug reports often include sufficient information for debugging techniques in order to localize [84], debug [61], and repair faults [30] that happened in the field. Therefore, together with specially crafted defect patterns (mined through systematic examination of real faults), such information can guide fault injection to target critical functionality, mimic real faulty behavior, and make realistic fault injection. Perhaps more importantly, the use of bug reports removes the need for knowledge of the targeted system or code.

Our method starts from the target project and a **bug report (BR)** written in natural language. It then applies **Information Retrieval (IR)**-based fault localization [84] in order to identify the relevant places where to inject faults. It then injects recurrent fault instances (fault patterns) that were manually crafted using a systematic analysis of frequent bug fixes, prioritized according to their position and type. This way our method performs fault injection, using realistic fault patterns, by targeting the features described by the bug reports. Moreover, by applying our method on many programs and BRs (injecting few bugs per BR), one gets fault pools to be used for test and fault tolerance assessment.

We implemented our approach in a system called iBiR and evaluated its ability to imitate 280 real faults. In particular, we evaluated (1) the semantic similarity of real and injected faults, (2) the coupling[1] relation between injected and real faults, and (3) the ability of the injected faults

[1]Injected faults couple with the real ones when injected faults are detected only by test cases that detect the real faults. This implies that the injected faults provide good indications on whether tests are capable of detecting the coupled faults.

to indicate test effectiveness (fault detection) when tested with different test suites. Our results show that iBiR manages to imitate the targeted faults, with a median semantic similarity value of 0.577, which is significantly higher than the 0.134 achieved by using traditional mutation testing when injecting the same number of faults.

Interestingly, we found that iBiR injects faults that couple with the real ones in around 36% of the targeted cases. This is achieved by injecting 100 faults per target (real) fault, and it is approximately nine times higher than the coupled mutants produced by mutation testing. Fault coupling is one of the most important testing properties [28, 57], here indicating that one can use the injected faults instead of the real ones.

Another key finding of our study is that the injected faults provide much better indication on test effectiveness (fault detection) than mutation testing as their detection ratios discriminate between actual failing and passing test suites, while mutant detection rates cannot. This implies that the use of iBiR yields more accurate results than the use of traditional mutation testing.

2 SCOPE AND MOTIVATION

iBiR aims at injecting realistic faults, i.e., faults imitating the behavior of previously reported ones, to be used for test and fault tolerance assessment. As such, it injects faults in a current stable (fixed) version of the same system where test techniques are assessed with respect to (1) fault revelation potential, in the case of test assessment, and (2) the reaction of the system under unexpected (faulty) behavior to support controlled studies. This means that we assume the existence of relatively stable projects with Fixed/Closed bug reports. In principle, iBiR could be use to guide testing toward open bug reports or to support the discovery of bugs that are similar to those reported. However, these two use cases regard the fault revelation ability of the fault injection campaigns (the test guidance provided by fault injection) and not the realistic fault injection problem (the ability of injecting faults to imitate the behavior of real ones) that we are aiming at. Therefore, we have left them open for future research.

2.1 Assessment of Testing Techniques

Fault injection is used extensively by researchers as a tool to evaluate the fault-revealing capability of automated test techniques such as automated test generation techniques. This approach was found to be used by approximately 19% of all software testing studies published in major SE conferences by a bibliometric analysis performed in 2016 [59]. This is because real and diverse bug datasets are hard to collect and make it hard to perform controlled studies as they usually result in faulty versions including single faults. Fault injection is thus a fast and convenient way to perform control studies since it avoids the costly and tedious work of creating fault datasets. In such cases, the realism of the injected faults is a major validity question that may impact the results of the experiments. Recent studies [62] have shown that conventional mutation testing doesn't perform well in this regard as it introduces many faults that are unrealistic. To deal with such cases, we develop iBiR and show that it injects more semantically similar faults than traditional mutation testing.

2.2 Fault Tolerance Assessment

Fault injection is also frequently used to evaluate the system's performance under faulty test executions. In such a case, the injected faults simulate the effects of real ones by performing arbitrary code changes everywhere. To this end, iBiR guides the injection toward specific error-prone targets/features and fault types. This is particularly important in order to improve the realism of the analysis. Interestingly, previous research on fault tolerance assessment [51] has shown that fault injection realism can be improved by appropriately controlling the locations and types of the injected faults. We therefore propose a way to do so by leveraging information from bug reports.

3 BACKGROUND

3.1 Fault Localization

Fault localization is the activity of identifying the suspected fault locations, which will be transformed to generate patches. Several automated fault localization techniques have been proposed [80], such as slice based [79], spectrum based [2], statistics based [37], mutation based [61], and so forth.

Fault localization techniques based on **Information Retrieval (IR)** [12, 18, 44, 67] exploit textual bug reports to identify code chunks relevant to the bug, without relying on test cases. IR-based fault localization tools extract tokens from the bug report to formulate a query to be matched with the collection of documents formed by the source code files [42, 65, 75, 77, 81, 84]. Then, they rank the documents based on their relevance to the query, such that source files ranked higher are more likely to contain the fault. Recently, automated program repair methods have been designed on top of IR-based fault localization [30]. They achieve comparable performance to methods using spectrum-based fault localization, yet without relying on the assumption that test cases are available.

We leverage IR-based fault localization to achieve a different goal; instead of localizing the reported bug, we aim at *injecting faults* at code locations that implement a functionality similar to the one described by the bug report.

3.2 Mutation Testing

Mutation testing is a popular fault-based testing technique [60]. It operates by inserting artificial faults into a program under test, thereby creating many different versions (named *mutants*) of the program. The artificial faults are injected through syntactic changes to all program locations in the original program, based on predefined rules named *mutation operators*. Such operators can, for instance, invert relational operators (e.g., replacing $\geq$ with $<$).

Mutants can be used to indicate the strengths of test suites, based on their ability to distinguish the mutants from the original program. If there exists a test case distinguishing the original program from a particular mutant, then the mutant is said to be *killed*. Then, we term a mutant to be "coupled" with respect to a particular fault if the test cases that kill it are a subset of the test cases that can also detect that fault (make the program fail by exerting the fault).

Previous research has shown that the choice of mutation operators and location can affect the fault-revealing ability of the produced mutants [5, 35]. Thus, it is important to select appropriate mutation testing strategies. Nevertheless, previous research has shown that random mutant sampling achieves comparable results with the mutation testing state of the art [9, 32], making the random mutant sampling a natural baseline to compare with.

Another issue with mutation testing regards its application cost. The problem stems from the vast number of faults that are injected, which need to be executed with large test suites, thereby requiring expensive computational resources [60]. Unfortunately, the mutant execution problem becomes intractable when test execution is expensive or the test suites involve system-level tests, thereby often limiting mutation testing application to unit level. This is a major problem when performing fault tolerance [51] or large-scale testing campaigns. Recent studies aim at reducing the computational demands of the mutant execution through a combination of static and dynamic metrics [82], but these methods cannot be applied for fault tolerance assessment and do not identify which mutants are realistic and which are not. Thus, it remains an open question to identify the few but realistic mutants.

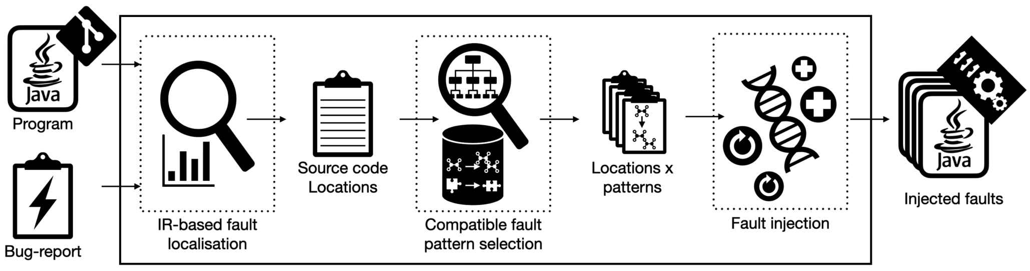


Fig. 1. The iBiR fault injection workflow.

We fill this gap by using bug-report-driven fault injection. In essence, we leverage IR-based fault localization techniques to identify the locations where fault injection should happen, i.e., locations relevant to the targeted functionality described in the bug report, and apply frequent fault patterns to produce mutants that behave similarly to real faults.

3.3 Fix Patterns

In automated program repair [22], a common way to generate patches is to apply fix patterns [26] (also named fix templates [38] or program transformation schemes [23]) in suspicious program locations (detected by fault localization). Patterns used in the literature [15, 23, 26, 31, 38, 38, 39, 46, 66] have been defined manually or automatically (mined from bug fix datasets).

Instead of fix patterns, we use *fault patterns* that are fix patterns inverted. Since fix patterns were designed using recurrent faults, their related fault patterns introduce them. This helps injecting faults that are similar to those described in the bug reports. iBiR inverts and uses the patterns implemented by *TBar* [40] as we detail in the following section.

4 APPROACH

We propose iBiR, the first fault injection approach that utilizes information extracted from bug reports to emulate real faults. A high-level view of the way iBiR works is shown in Figure 1 and a step-by-step overview of IBIR's approach is illustrated in Algorithm 1. Our approach takes as input (1) the source code of the program of interest and (2) a bug report of that program, written in natural language. The objective is to inject artificial faults in the program (one by one, creating multiple faulty versions of the program) that imitate the original bug. To do so, iBiR proceeds in three steps.

First step: iBiR identifies relevant locations to inject the faults. It applies IR-based fault localization to determine, from the bug report, the code locations (statements) that are likely to be relevant to the target fault. These locations are ranked according to their likelihood to be the feature described by the bug report, and hence are relevant to inject faults.

Second step: iBiR applies fault patterns on the identified code locations. We build our patterns by inverting fix patterns used in automated program repair approaches [40]. Our intuition is that, since fix patterns are used to fix bugs, inverted patterns may introduce a fault similar to the original bug. For each location, we apply only patterns that are syntactically compatible with the code location. This step yields a set of faults to inject, i.e., pairs composed of a location and a pattern.

Third step: our method ranks the location-pattern pairs w.r.t. the location likelihood and priority order of the patterns. Then iBiR takes each pair (in order) and applies the pattern to the location, injecting a fault in the program. We repeat the process until the desired number of injected faults has been produced or until all location-pattern pairs have been considered.

ALGORITHM 1: IBIR approach algorithm

```
Require: $bugReport, projectRepository, numberOfFaults$
 1: $patterns[] \leftarrow$ loadListOfPatterns()
 2: $patches[] \leftarrow []$
 3: $result[] \leftarrow []$
 4: $rankedSuspeciousFiles[] \leftarrow$ fileLevelIRFL($bugReport, projectRepository$)
 5: $first20RankedSuspeciousFiles[] \leftarrow$ head($rankedSuspeciousFiles, 20$)
 6: $rankedSuspeciousStatements[] \leftarrow$ statementLevelIRFL($bugReport, first20RankedSuspeciousFiles[]$)
 7: for $statement$ in $rankedSuspeciousStatements[]$ do
 8:   $fileAstTree \leftarrow$ loadAstTree($statement.containingFile$)
 9:   $statementNodes[] \leftarrow$ parseTree($tree, statement$)
10:   for $astNode$ in $statementNodes[]$ do
11:     for $pattern$ in $patterns[]$ do
12:       if patternIsAppliableOnNode($pattern, astNode$) then
13:         $patch \leftarrow$ createPatch($pattern, astNode, fileAstTree$)
14:         add($patch, patches[]$)
15:       end if
16:     end for
17:   end for
18: end for
19: for $patch$ in $patches[]$ do
20:   $faultyVersion \leftarrow$ apply($patch, projectRepository$)
21:   if isCompilable($faultyVersion$) then
22:     add($patch, result[]$)
23:   end if
24:   if $numberOfFaults ==$ length($result[]$) then
25:     return $result[]$
26:   end if
27: end for
28: return $result[]$
```

4.1 Bug-report-driven Fault Localization

IR-based fault localisation (IRFL) [63, 74] leverages potential similarity between the terms used in a bug report and the program source code to identify relevant buggy code locations. It typically starts by extracting tokens from a given bug report to formulate a *query* to be matched in a search space of *documents* formed by the collections of source code files and indexed through tokens extracted from source code [42, 65, 75, 77, 78, 84]. IRFL approaches then rank the documents based on a probability of relevance. Top-ranked files are likely to contain the buggy code.

We follow the same principle to identify promising locations where to inject realistic faults. We rely on the information contained in the bug report to localize the code location with the highest similarity score. Most IRFL techniques have focused on file-level localization, which is too coarse-grained for our purpose of fault injecting. Thus, we rather use a statement-level IRFL approach that has been successfully applied to support program repair [30].

It is to be noted that, contrary to program repair, we do not aim to identify the exact bug location. We are rather interested in locations that allow injecting realistic faults (similar to the bug). This means that IRFL may pinpoint multiple locations of interest for fault injection even if those were not buggy code locations.

To identify fault injection locations that are relevant to the targeted bug report, we leverage an existing IRFL tool that was originally developed as part of the iFixR [30] tool. The IRFL works by matching words of a bug report with source code file(s) using 17 features. These features are

Table 1. IR Features Collected from Bug Reports and Source Code Files

Bug Report Features	
Feature	**description**
summary	The summary/title part of the bug report
description	The description part of the bug report
rawBugReport	The whole bug report as in textual form
stackTraces	The stack traces in the bug report
codeElements	Code snippets in the bug reports
summaryHints	Code-related terms in summary
descriptionHints	Code-related terms found by parsing description text
Source Code Features	
Feature	**description**
packageNames	The parsed package names of the source code files
classNames	The parsed class names of the source code files
methodNames	The parsed method names of the source code files
methodInvocations	The parsed method invocation of the source code files
formalParameters	The parsed formal parameters of the source code files
memberReferences	The parsed member references of the source code files
documentation	The parsed class names of the source code files
rawSource	Source file as a text
hunks	The hunks from the commits on the file
commitLogs	The commit logs of the file

extracted from the bug report (7 features) and the source code git repository (10 features) and are listed in Table 1.

For every feature, the tokenizer applies a lexical analysis where (1) it extracts tokens from the retrieved text; (2) then drops stopwords to reduce the noise, i.e., caused by the programming language keywords; and (3) applies stemming on all tokens to create homogeneity with the root of the token. The tokens are extracted by considering both white space and source-code-specific separators, such as punctuation and camel case splitting; i.e., *calculateMaximum* is split to *calculate* and *Maximum*. The tokens are then checked against the WordNet [16] dictionary to discard all unknown ones. An additional sanity check is then applied to detect stack-traces and source code elements using specific regular expressions.

The IRFL calculates then the similarity coefficient (*Cosine* [64]) between the bug report and a source code file using a **revised Vector Space Model (rVSM)** [85] based on the occurrence frequency of the extracted tokens in the preprocessing tokenization step (the vectors are calculated using $tf - idf$ [47]).

Next, an ensemble of classification models provided by D&C [29] was used in order to rank the source code files according to their suspiciousness. This ensemble takes as input the calculated 7×10 weights of all pairs <bug report, source code file> and outputs their averaged prediction results. This ensemble was used as it has been shown to work well on a diverse set of bug reports [29] since every classifier of the ensemble model was trained on a different set of data.

In a last step, as iFixR [30], the IRFL localizes suspicious statements from the 20 most suspicious files based on their rVSM cosine-similarity [64] with the given bug report (the vectors are calculated using $tf - idf$ [47]) and outputs these statements in a list of statements ranked according to their suspiciousness. Further details on the IRFL can be found in the D&C work [29] and our implementation [1].

Table 2. iBIr Fault Injection Patterns

Pattern Context Category	Bug Injection Pattern	Example Input	Example Output
Insert Statement	Insert a method call, before or after the localized statement.	someMethod(expression);	someMethod(expression); method(expression);
	Insert a return statement, before or after the localized statement.	statement;	**statement; return VALUE;**
	Wrap a statement with a try-catch.	statement;	try{ statement; } catch (Exception e){ ... }
	Insert an if checker: wrap a statement with an if block.	statement;	if (conditional_exp) { statement; }
Mutate Class Instance Creation	Replace an instance creation call by a cast of the super.clone() method call.	... new T();	... (T) super.clone();
Mutate Conditional Expression	Remove a conditional expression.	condExp1 && condExp2	condExp1
	Insert a conditional expression.	condExp1	condExp1 && condExp2
	Change the conditional operator.	condExp1 && condExp2	condExp1 || condExp2
Mutate Data Type	Change the declaration type of a variable.	T1 var ...;	T2 var ...;
	Change the casting type of an expression.	... (T1) expression ...;	... (T2) expression ...;
Mutate Float or Double Division	Remove a float or a double cast from the divisor.	... dividend / (float) divisor ...; ... intVarExp / 10d ...;	... dividend / divisor ...; ... intVarExp / 10 ...;
	Remove a float or a double cast from the dividend.	... (float) dividend / divisor ...; ... 1.0 / var ...;	... dividend / divisor ...; ... 1 / var ...;
	Replace float or double multiplication by an int division.	... (1.0 / divisor) * dividend ... ... 0.5 * intVarExp ...;	... dividend / divisor ...; ... intVarExp / 2 ...;
Mutate Literal Expression	Change Boolean, number, or string literals in a statement by another literal or expression of the same type.	... string_literal1 ... ... int_literal ...	... string_literal2 ... ... int_expression ...
Mutate Method Invocation	Replace a method call by another one.	... method1(args) ...	... method(args) ...
	Replace a method call argument by another one.	... method(arg1, arg2) ...	... method(arg1, arg3) ...
	Remove a method call argument.	... method(arg1, arg2) ...	... method(arg1) ...
	Add an argument to a method call.	... method(arg1) ...	... method(arg1, arg2) ...
Mutate Return Statement	Replace a return expression by another one.	return expr1;	return exp2;
Mutate Variable	Replace a variable by another variable or an expression of the same type.	... var1 ... ... var1 ...	... var2 ... ... exp ...
Move Statement	Move a statement to another position.	statement; ...	... statement;
Remove Statement	Remove a statement.	statement; ...	...
	Remove a method.	method(args){ statement; }	...
Mutate Operators	Replace an Arithmetic operator.	... a + b ...	... a - b ...
	Replace an Assignment operator.	... c += b ...	... c -= b ...
	Replace a Relational operator.	... a < b ...	... a > b ...
	Replace a Conditional operator.	... a && b ...	... a || b ...
	Replace a Bitwise or a Bit Shift operator.	... a & b ...	... a | b ...
	Replace a Unary operator.	a++	a--
	Change arithmetic operations order.	a + b * c	c + b * a

4.2 Fault Patterns

We start from the fix patterns developed in TBar [40], a state-of-the-art pattern-based program repair tool. Any pattern is described by a context, i.e., an AST node type to which the pattern applies, and a recipe, a syntactical modification to be performed similar to program repair techniques [76]. For each pattern, we define a related fault injection pattern that represents the inverse of that pattern. For instance, inverting the fix pattern that consists of adding an arbitrary statement yields a *remove statement* fault pattern. Interestingly, some fix patterns are symmetric in the sense that their inverse pattern is also a fix pattern, e.g., inverting a Boolean connector. These patterns can thus be used for both bug fixing and fault injection. Table 2 enumerates the resulting set of fault injection patterns used by our approach.

Given a location (code statement) to inject a fault into, we identify the patterns that can be applied to the statement. To do so, our method starts from the AST node of the statement and visits it exhaustively, in a breadth-first manner. Each time it meets an AST node that matches the context of a fault pattern, it memorizes the node and the pattern for later application. Then the method continues until it has visited all AST nodes under the statement node. This way, we enumerate all possible applications of all fault patterns onto the location.

Since more than one pattern may apply to a given location, we prioritize them by leveraging heuristic priority rules previously defined in automated program repair methods (these were inferred from real-world bug occurrences [40]). This means that every fault injection pattern gets the priority order of its inverse fix pattern.

4.3 Fault Injection

The last step consists of applying, one by one, the fault patterns to inject faults at the program locations identified by IRFL. Locations of higher ranking are considered first. Within a location, pattern applications are ordered based on the pattern priority. By applying a pattern to a corresponding AST node of the location, we inject a fault within the program before recompiling it. If the program does not compile, we discard the fault and restart with the next one. We continue the process until it reaches the desired number of (compilable) injected faults or all locations and patterns have been considered.

4.4 Demonstration Example

Figures 2 and 3 illustrate the execution steps of iBiR when injecting faults in commons-math project, based on the content of the bug report MATH-329.[2]

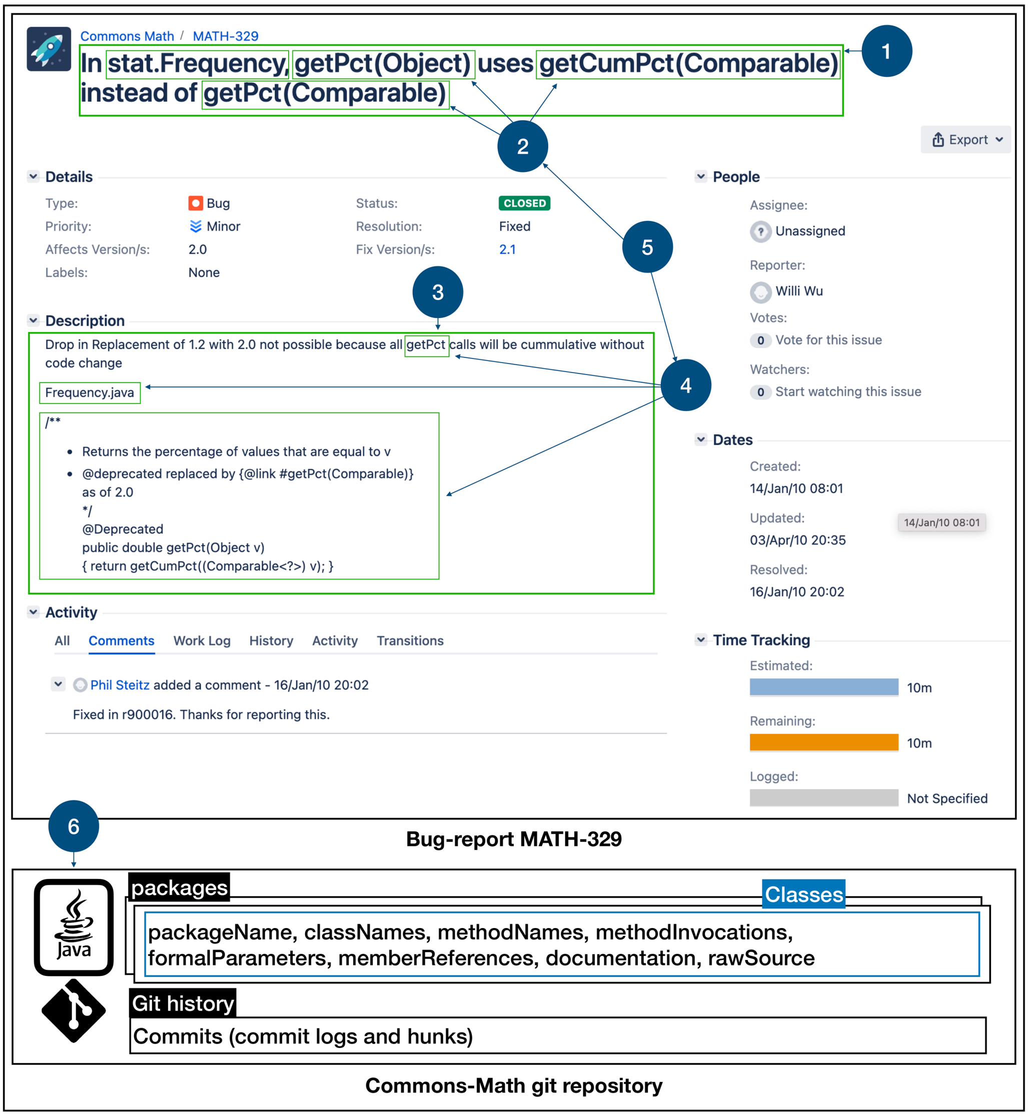
iBiR starts by parsing the bug report and extracting its relevant information: the summary (1), the summary hints (2), the description (3), the description hints (4), code elements (5), and the raw bug report. This example bug report does not contain any stack-trace as the corresponding bug causes a misbehavior but does not trigger any crash or throw any exception.

iBiR loads also all the required information from the project's repository (6) and then uses all of these features to find the code locations that are the most likely related to the input bug report. This search happens in two steps—file-level then statement-level localization—and ends by the output of a sorted list of source-code lines (7), as detailed in Section 4.1.

iBiR parses these lines one by one starting with the highest rank. In this example, the first rank is attributed to the line number `303` of the file `src/main/java/org/apache/commons/math/stat/Frequency.java` (8), which corresponds to a `return` statement that invokes the method `getPct` with a variable `v`, which is cast to the type `Comparable`. iBiR selects all compatible fault patterns with this statement's AST and applies them one by one on the source code, inducing multiple faults. In Figure 3 we illustrate the modified source code corresponding to five faults injected in the line `303` of the `Frequency.java` file (9): Faults 1 and 2 are injected by invoking respectively the methods `getCumPct` and `getCumFreq` instead of `getPct`. In fault 3, the method `getPct` is invoked with the field `this.freqTable` as variable instead of `v`. Faults 4 and 5 are injected by inserting additional method calls before the `return` statement, respectively `addValue(v);` and `clear();`.

iBiR continues parsing the sorted source code locations by the IRFL until all of them are treated or the requested number of faults has been injected.

[2]Bug report link: https://issues.apache.org/jira/browse/MATH-329.



Commons Math / MATH-329

In stat.Frequency, getPct(Object) uses getCumPct(Comparable) instead of getPct(Comparable)

Details

Type:	Bug	Status:	CLOSED
Priority:	Minor	Resolution:	Fixed
Affects Version/s:	2.0	Fix Version/s:	2.1
Labels:	None		

Description

Drop in Replacement of 1.2 with 2.0 not possible because all getPct calls will be cummulative without code change

Frequency.java

```
/**
 • Returns the percentage of values that are equal to v
 • @deprecated replaced by {@link #getPct(Comparable)}
   as of 2.0
   */
   @Deprecated
   public double getPct(Object v)
   { return getCumPct((Comparable<?>) v); }
```

Activity

All Comments Work Log History Activity Transitions

Phil Steitz added a comment - 16/Jan/10 20:02

Fixed in r900016. Thanks for reporting this.

People

Assignee: Unassigned

Reporter: Willi Wu

Votes: 0 Vote for this issue

Watchers: 0 Start watching this issue

Dates

Created: 14/Jan/10 08:01

Updated: 03/Apr/10 20:35 14/Jan/10 08:01

Resolved: 16/Jan/10 20:02

Time Tracking

Estimated: 10m

Remaining: 10m

Logged: Not Specified

Bug-report MATH-329

packages

Classes

packageName, classNames, methodNames, methodInvocations, formalParameters, memberReferences, documentation, rawSource

Git history

Commits (commit logs and hunks)

Commons-Math git repository

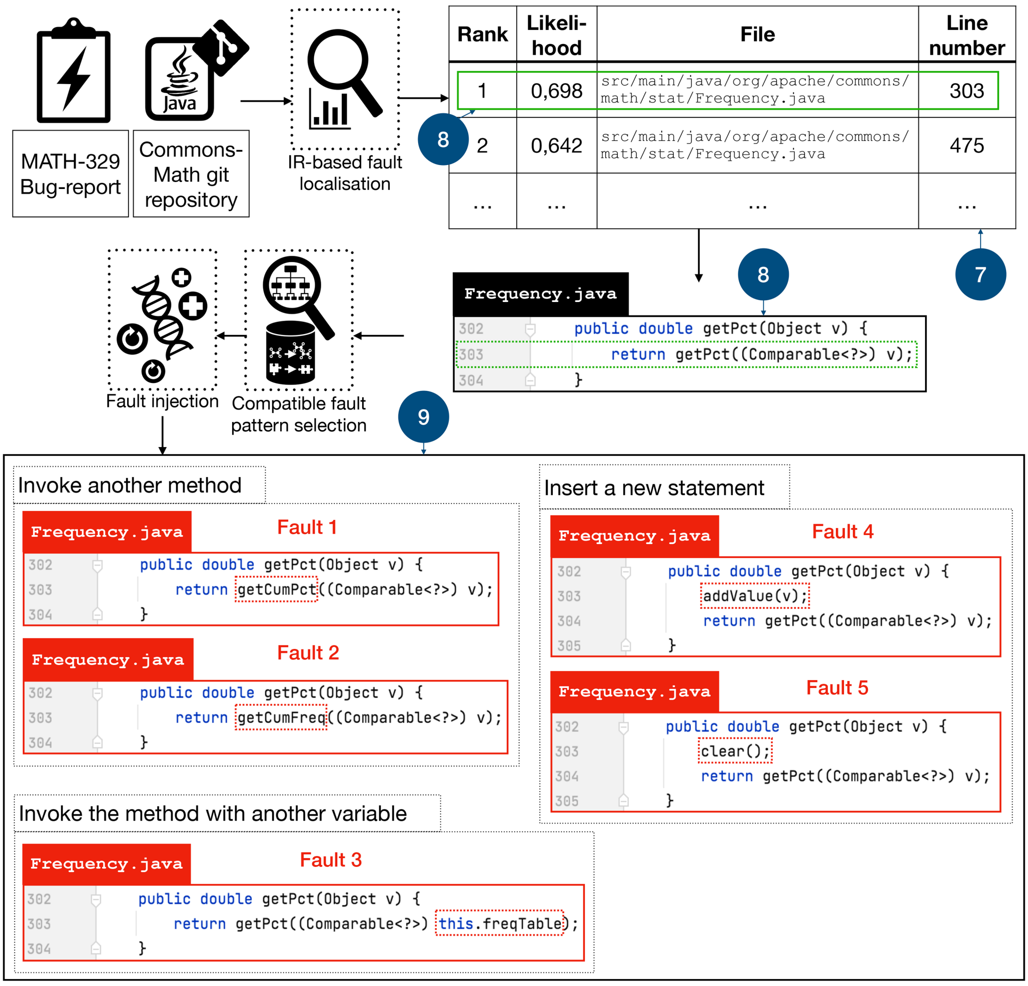
Fig. 2. Example of IBIR's input: the bug report MATH-329 (1, the summary; 2, the summary hints; 3, the description; 4, the description hints; 5, code elements) and the Commons-Math git repository (6).

5 RESEARCH QUESTIONS

Our approach aims at injecting faults that imitate real ones by leveraging the information included in bug reports. Therefore, a natural question to ask is how well IBIR's faults imitate the targeted (real) ones. Thus, we ask:

RQ1 *(Imitating bugs):* Are the IBIR faults capable of emulating, in terms of semantic similarity, the targeted (real) ones? How do they compare with mutation testing?

To answer this question, we check whether any of the injected faults imitate well the targeted ones. Following the recommendations from the mutation testing literature [62], we approximate


MATH-329 Bug-report
Commons-Math git repository
Java
IR-based fault localisation
8
7
9
Fault injection
Compatible fault pattern selection
Invoke another method
Invoke the method with another variable
Insert a new statement
Fault 1
Fault 2
Fault 3
Fault 4
Fault 5

Rank	Likelihood	File	Line number
1	0,698	src/main/java/org/apache/commons/math/stat/Frequency.java	303
2	0,642	src/main/java/org/apache/commons/math/stat/Frequency.java	475
...	...	...	...

Frequency.java
```java
public double getPct(Object v) {
    return getPct((Comparable<?>) v);
}
```

Frequency.java — Fault 1
```java
public double getPct(Object v) {
    return getCumPct((Comparable<?>) v);
}
```

Frequency.java — Fault 2
```java
public double getPct(Object v) {
    return getCumFreq((Comparable<?>) v);
}
```

Frequency.java — Fault 3
```java
public double getPct(Object v) {
    return getPct((Comparable<?>) this.freqTable);
}
```

Frequency.java — Fault 4
```java
public double getPct(Object v) {
    addValue(v);
    return getPct((Comparable<?>) v);
}
```

Frequency.java — Fault 5
```java
public double getPct(Object v) {
    clear();
    return getPct((Comparable<?>) v);
}
```

Fig. 3. Example of iBiR's execution on the bug report MATH-329: the IRFL extracts tokens from the bug report and the projects repository. Then, it outputs a list of statements ranked by their suspiciousness (7, the two first ranked statements by iBiR). The mutator loads every statement in this list, parses its AST, selects the applicable patterns, and applies them one by one to inject faults (8, the statement with the highest suspiciousness; 9, faults injected when processing the first statement).

the program behavior through the project test suites and compare the behavior similarity of the test cases w.r.t. their pass and failing status using the Ochiai similarity coefficient. This is a typical way of computing the semantic similarity of mutants and faults in mutation-based fault localization [50, 61].

We then compare these results with the mutation testing ones by injecting mutants using the standard operators employed by mutation testing tools [28] and measuring their semantic similarity with the targeted faults. To make a fair comparison, we inject the same number of faults per target. For iBiR we selected the top-ranked mutants, while for mutation testing we randomly sampled mutants across the entire project code-base. Random mutant sampling forms our baseline since it performs comparably to the alternative mutant selection methods [9, 32]. Also, since we are interested in the relative differences between the injected fault sets, we repeat our experiments multiple times using the same number of faults (mutants).

Our approach identifies the locations where bugs should be injected through an IR-based fault localization method. This may give significant advantages when applied at the project level, but these may not carry on individual classes. Such class-level granularity may be well suited for some test evaluation tasks, such as automatic test generation [20]. To account for this, we performed mutation testing (using the traditional mutation operators) at the targeted classes (classes where the faults were fixed). To make a fair comparison we also restricted iBiR to the same classes and compared the same number of mutants. This leads us to the following question:

RQ2 *(Comparison at the target class):* How does iBiR compare with mutation testing, in terms of semantic similarity, when restricted to particular classes?

We answer this question by injecting faults in only the target classes using the iBiR bug patterns and the traditional mutation operators. Then we compare the two approaches the same way as we did in RQ1.

Up to this point, the answers to the posed questions provide evidence that using our approach yields mutants that are semantically similar to the targeted bugs. Although this is important and demonstrates the potential of our approach, it does not necessarily mean that the injected faults are strongly coupled with the real ones.[3] Mutant and fault coupling is an important property for mutants that significantly helps testing [25]. Therefore, we seek to investigate:

RQ3 *(Mutant and fault coupling):* How does iBiR compare with mutation testing with respect to mutant and fault coupling?

To answer this question, we check whether the faults that we inject are detected only by the failing tests, i.e., only by the tests that also reveal the target fault. Compared to similarity metrics, this coupling relation is stricter and stronger.

After answering the above questions, we turn our attention to the actual use of mutants in test effectiveness evaluations. Therefore, we are interested in checking the correlations between the failure rates of the sets of the injected faults we introduce and the real ones. To this end, we ask:

RQ4 *(Failure estimates):* Are the injected faults leading to failure estimates that are representative of the real ones? How do these estimates compare with mutation testing?

The difference of RQ4 from the other RQs is that in RQ4, a set of injected faults is evaluated, while in the previous RQs, only isolated mutant instances are evaluated.

6 EXPERIMENTAL SETUP

6.1 Dataset and Benchmark

To evaluate iBiR we needed a set of benchmark programs, faults, and bug reports. We decided to use Defects4J [24] since it is a benchmark that includes real-world bugs and it is quite popular in software engineering literature.

6.1.1 Linking the Bugs with Their Related Reports. We used the bug report to revision-id (commit) mapping provided by the Defects4J dataset. Unfortunately, none of the provided revision-ids for the projects Lang and Math were pointing to the actual git repositories, as the projects have been migrated into GitHub but the revision-ids didn't get updated in the dataset. So for these two projects, to identify which bug report describes a given bug in Defects4J, we followed the same

[3]Mutants are coupled with real faults if they are killed only by test cases that also reveal the real faults.

process as in the study of Koyuncu et al. [30]. We used the bug linking strategies that are implemented in the Jira issue tracking software and used the approach of Fischer et al. [17] and Thomas et al. [69] to map the sought bugs with the corresponding reports. Precisely, we crawled the relevant bug reports and checked their links. We selected bug reports that were tagged as "BUG" and marked as "RESOLVED" or "FIXED" and have a "CLOSED" status. Then we searched the commit logs to identify related **identifiers (IDs)** that link the commits with the corresponding bug.

Additionally, because of the limitations in our current IRFL implementation, we included only the projects that are using Jira as issue tracking software.

Our resulting bug dataset included the 316 faults of Defect4J related to the Cli (39), Codec (18), Collections (4), Compress (47), Csv (16), JxPath (22), Lang (64), and Math (106) projects. We discarded 36 defects because they were sharing the same bug report and we could not map the correct one with its related issue, issues with the buggy program versions such as missing files from the repository, or execution issues at the reporting time. This leaves us with a total of 280 faults.

6.2 Experimental Procedure

To compare the fault injection techniques we need to set a common basis for comparison. We set this basis as the number of injected faults since it forms a standard cost metric [53] that puts the studied methods under the same cost level. We used sets of 5, 10, 30, and 100 injected faults since our aim is to equip researchers with few representative faults, per targeted fault, in order to reach reasonable execution demands. To reduce the arbitrariness due to the stochastic nature of mutation testing, we reproduced the injection 15 times, and then we sorted the executions by their average Ochiai coefficient (for every bug separately) and we reported the mean execution. On the other hand, we run iBiR only once as its approach does not depend on random decisions.

To measure how well the injected faults imitate the real ones (answer RQ1 and RQ2) we use a semantic similarity metric (Ochiai coefficient) between the test failures on the injected and real (targeted) faults. Precisely, let fTS_M and fTS_B be the sets of failing tests when executing a test suite TS correspondingly on a mutant M and a buggy project B; the Ochiai coefficient is 0 if any of fTS_M or fTS_B is empty, or else is calculated as $Ochiai(M, B) = \frac{|fTS_M \cap fTS_B|}{\sqrt{|fTS_M| . |fTS_B|}}$, where $|set|$ denotes the set size. In our study, as we're executing the fixed-version test suites provided by Defects4J, every targeted bug breaks at least one test, and thus, fTS_B is never empty. This coefficient quantifies the similarity level of the program behaviors exercised by the test suites and is often used in mutation testing literature [62]. The metric takes values in the range [0, 1], with 0 indicating complete difference and 1 exact match. We treated the injected faults that were not detected by any of the test suites as equivalent mutants [6, 58]. This choice does not affect our results since we approximate the program behaviors through the projects test suites; i.e., they are never killed.

To measure whether the injected faults couple with the existing ones (answer RQ3), we followed the process suggested by Just et al. [25] and identified whether there were any injected faults that were killed by at least one failing test (test that detects the real fault) and not by any passing test (test that does not detect the real fault). In RQ4 we randomly sampled 50 test suites, random subsets of the accompanied test suites, that included between 10% and 30% test cases of the original test suite (provided by Defects4J). Thus, we ensure that the selected samples (1) are smaller than the original test suite, (2) have different sizes, and (3) have different ratios of killing the mutants and detecting the targeted bug. Then we recorded the ratios of the injected faults that are detected when injecting 5, 10, 30, and 100 faults. We also recorded binary variables indicating whether or not each test suite detects the targeted fault. This process simulates cases where test suites of different strengths are compared. Based on these data, we computed two statistical correlation coefficients, the Kendall and Pearson.

To further validate whether the two approaches provide sufficient indicators on the effectiveness of the test suites, we check whether the detection ratios of the injected faults are statistically higher when test suites detect the targeted faults than when they do not.

To reduce the influence of stochastic effects we used the Wilcoxon test with a significance level of 0.05. This helped in deciding whether the differences we observe can be characterized as statistically significant. Statistical significance does not imply sizable differences, and thus, we also used the Vargha Delaney effect size $\hat{A}_{12}$ [71]. In essence, the $\hat{A}_{12}$ values quantify the level of the differences. For instance, a value $\hat{A}_{12} = 0.5$ can be interpreted as a tendency of equal value of the two samples. $\hat{A}_{12} > 0.5$ suggests that the first set has higher values, while $\hat{A}_{12} < 0.5$ suggests the opposite.

6.3 Implementation

To perform our experiments, we implemented iBiR's approach as described in Section 4: we have used the IRFL implementation proposed in iFixR [30] and implemented the mutator component that is responsible for injecting faults in specific locations, as a java standalone application. Second, for the mutation testing, denoted as "Mutation" in our experiments, we used randomly sampled mutants from those produced by typical mutation operators, coming from the mutation testing literature. In particular, we implemented the muJava intra-method mutation operators [43], which are the most frequently used [28]. Third, to reduce the noise from stillborn mutants, i.e., mutants that do not compile, we discarded without taking into any consideration, i.e., prior to our experiment, every mutant that did not compile or whose execution with the test suite exceeded a timeout of 5 minutes. Fourth, when answering the RQ1, we found out that there were many cases where iBiR injected fewer than 100 faults. To perform a fair comparison, we discarded these cases (for both approaches). This means that we always report results where both studied approaches manage to inject the same number of faults.

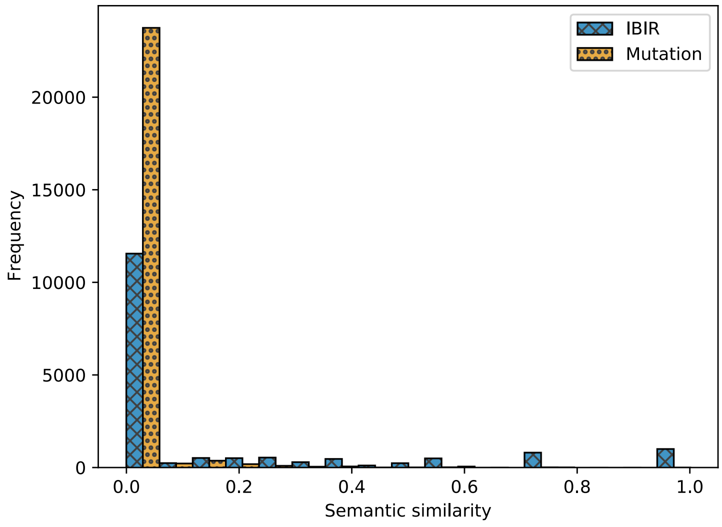
7 RESULTS

7.1 RQ1: Semantic Similarity between iBiR Injections and the Targeted Real Faults

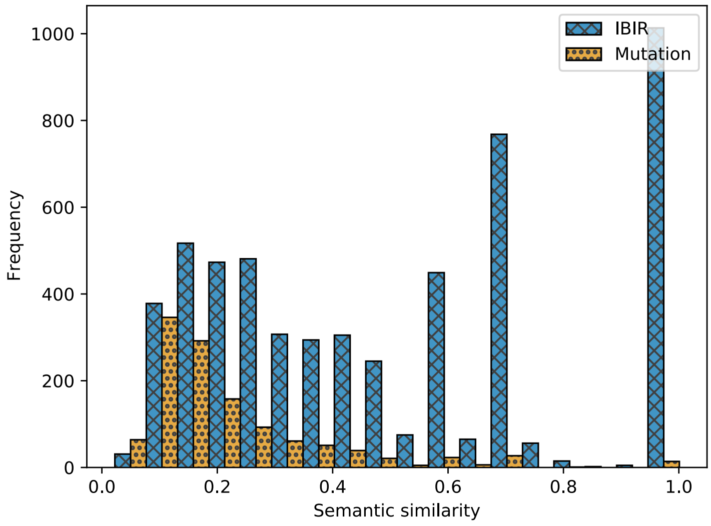
To check whether the injected faults imitate well the targeted ones, we measured their behavior (semantic) similarity w.r.t. the project test suites (please refer to Section 6 for details). Figure 4 shows the distribution of the similarity coefficient values that were recorded in our study. As can be seen, iBiR injects hundreds of faults that are similar to real ones, whereas mutation testing (denoted as "Mutation" in Figure 4) did not manage to generate any. At the same time, as typically happens in mutation testing [62], a large number of injected faults have low similarity. This is evident in our data, where mutations have 0 similarity.

To investigate whether iBiR successfully injects any fault that is similar (semantically) to the targeted ones, we collected the best similarity coefficients, per targeted fault, when injecting 5, 10, 30, and 100 faults. Figure 5 shows the distribution of these results. For more than half of the targeted faults, iBiR yields a best similarity value higher than 0.5 when injecting 100 faults, indicating that iBiR's faults imitate relatively well the targeted ones. We also observe that in many faults the best similarity values are above 0 by injecting just 10 faults. This is important since it indicates that iBiR successfully identifies relevant locations for fault injection.

To establish a baseline and better understand the value of iBiR, we need to contrast iBiR's performance with that of mutation testing when injecting the same number of faults. Mutation testing forms the current SoA of fault injection and thus a related baseline. As can be seen from Figure 5, the similarity values of mutation testing are significantly lower than those of iBiR.



(a) All injected faults.



(b) Faults with an Ochiai coefficient higher than zero.

Fig. 4. Distribution of semantic similarities of 100 injected faults per targeted (real) fault.

iBiR injects faults that resemble those described in Bug Reports. iBiR injects a fault that imitates the real targeted one significantly better than traditional mutation testing.

Figure 6 shows the distribution of the semantic similarities, between real and injected faults, when injecting 5, 10, 30, and 100 faults. As can be seen from the boxplots, the trend is that a large portion of faults injected by iBiR have positive similarity scores with the targeted ones.

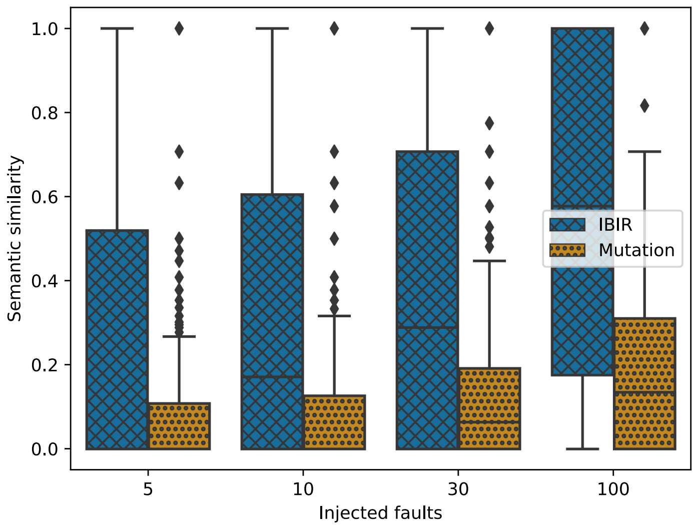


Fig. 5. Semantic similarity per targeted (real) fault, top values. IBIR injects faults with higher similarity coefficients than mutation testing.

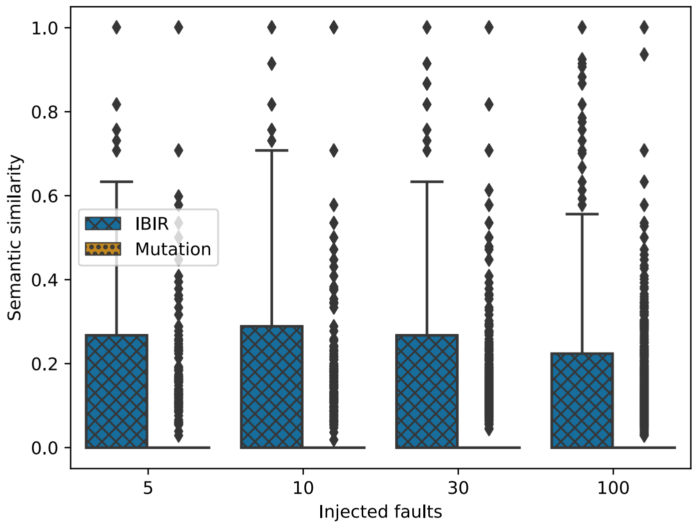


Fig. 6. Semantic similarity of all injected faults. IBIR injects faults with higher similarity coefficients than mutation testing.

Interestingly, in mutation testing, only outliers have their similarity above 0. In particular, mutation testing injected faults with similarity values higher than 0 in 87, 112, 145, and 189 of the targeted faults (when injecting 5, 10, 30, and 100 faults), while IBIR injected in 130, 156, 190, and 226 of the targeted faults, respectively.

To validate this finding, we performed a statistical test (Wilcoxon paired test) on the data of both Figures 5 and 6 to check for significant differences. Our results showed that the differences are significant, indicating the low probability of this effect to be happening by chance. The size of

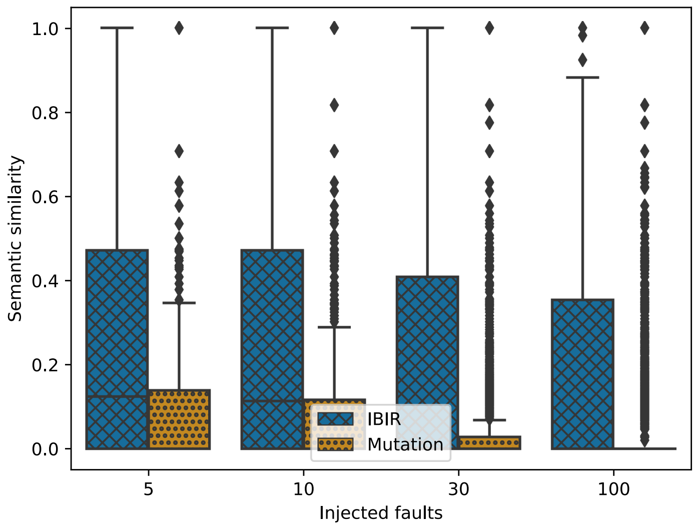


Fig. 7. Semantic similarity of injected faults at particular classes. iBiR injects faults with higher similarity coefficients than mutation testing (at class-level granularity).

the difference is also big, with iBiR yielding $\hat{A}_{12}$ values between 0.64 and 0.68, indicating that iBiR injects faults with higher semantic similarity to real ones in the great majority of the cases. Due to the many cases with 0 similarity values, the average similarity value of iBiR's faults is 0.163, while for mutation it is 0.010, indicating the superiority of iBiR.

iBiR injects faults that better resemble real faults than traditional mutation testing in 64% to 68% of the cases.

7.2 RQ2: iBiR vs. Mutation Testing at Particular Classes

To check the performance of iBiR at the class level of granularity we repeated our analysis by discarding, from our priority lists, every mutant that is not located on the targeted classes, i.e., classes where the targeted faults have been fixed. Figure 7 shows the distribution of the semantic similarities when injecting 5, 10, 30, and 100 faults at a particular class. As expected, mutation testing scores are higher than those presented before, but still mutation testing falls behind.

To validate this finding, we performed a statistical test and found that the differences are significant. The size of the difference is between 0.62 and 0.65, meaning that iBiR scores more than 60% times higher than mutation testing. The average similarity values of the iBiR faults is 0.217, while for mutation it is 0.066, indicating that iBiR is better.

iBiR outperforms traditional mutation testing, imitating real faults, even when restricted to a particular (target) class. The difference is significant, with iBiR scoring more than 60% of the time higher than mutation testing.

7.3 RQ3: Fault Coupling

The coupling between the injected and the real faults forms a fundamental assumption of the fault-based testing approaches [24]. An injected fault is coupled to a real one when a test case that

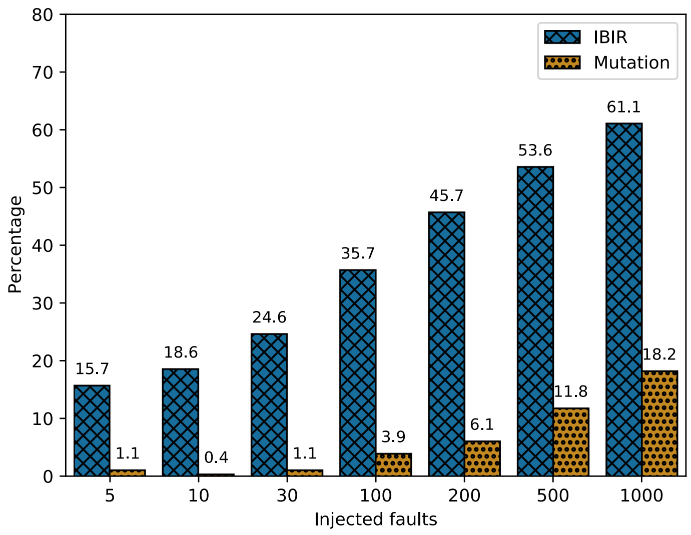


Fig. 8. Percentage of real faults that are coupled to injected ones when injecting 5 to 1,000 faults.

reveals the injected fault also reveals the real fault [24]. This implies that revealing these coupled injected faults results in revealing potential real ones. We therefore check this property in the faults we inject and contrast it with the baseline mutation testing approach.

Figure 8 shows the percentage of targeted faults where there is at least one injected fault that is coupled to a real one. This is shown for the scenarios where 5, 10, 30, and 100 faults per target are injected. As we can see from these data, iBiR injects coupled faults for approximately 16% of the target faults when it aims at injecting 5 faults. This percentage increases to 36% when the number of injected faults is increased to 100.

Perhaps surprisingly, mutation testing did not perform well (it injected coupled faults for around 4% of the target, when injecting 100 faults per target). These results differ from those reported by previous research [25, 62], because (1) previous research only injected faults at the faulty classes and not the entire project and (2) previous research injected all possible mutant instances and not 100 as we do.

iBiR injects coupled faults for approximately 16% to 36% of the cases, while mutation testing does it in around 4%. This is achieved by injecting 5 to 100 faults.

7.4 RQ4: Fault Detection Estimates

The results presented so far provide evidence that some of the injected faults imitate well the targeted ones, though the question of whether the injections provide representative results of real faults remains, especially since we observe a large number of faults with low similarity values. Therefore, we check the correlations between the failure rates of the sets of injected faults and the real faults when executed with different test suites (please refer to Section 6 for details).

Figure 9 shows the distribution of the correlation coefficients when injecting different numbers of faults. Interestingly, the results on both figures show a trend in favor of iBiR. This difference is statistically significant, shown by a Wilcoxon test, with an effect size of approximately 0.6. Table 3 records the effect size values, $\hat{A}_{12}$, for the examined strategies. In essence, these effect sizes mean that iBiR outperforms the mutant injection in 60% of the cases, suggesting that iBiR could be a

Table 3. Vargha and Deianey $\hat{A}_{12}$ (iBiR vs. Mutation) of Kendall and Pearson Correlation Coefficients

Number of Injected Faults	**5**	**10**	**30**	**100**
Kendall	0.605	0.620	0.681	0.655
Pearson	0.580	0.612	0.627	0.652

much better choice than mutation testing, especially in cases of large test suites with expensive test executions.

To further validate whether iBiR's faults provide good indicators (estimates) of test effectiveness (fault detection) we split our test suites between those that detect the targeted faults and those that do not. We then tested whether detection ratios of the injected faults in the test suite group that detects the real faults are significantly (statistically) higher than those in the group that does not detect it. In case this happens, we have evidence that our injected faults favor test suites capable of detecting real faults. This is important when comparing test generation techniques, where the aim is to identify the most effective (at detecting faults) technique.

Figure 10 records the number of faults where (real) fault-detecting test suites detect a statistically higher number of injected faults than those test suites that do not detect them. As can be seen by these results, iBiR has a big difference from mutation; i.e., it distinguishes between passing and failing test suites in 126 faults, while mutation does so in 55 faults. We also measured the Vargha and Delaney $\hat{A}_{12}$ effect size values on the same data, recorded in Figure 11. Of course it does not make sense to contrast insignificant cases, so we only performed that on the results where iBiR has a statistically significant difference. Interestingly, big differences are recorded (in approximately 80% of the cases) in favor of our approach.

iBiR injects faults that provided better fault detection estimates than traditional mutation testing in approximately 80% of the cases.

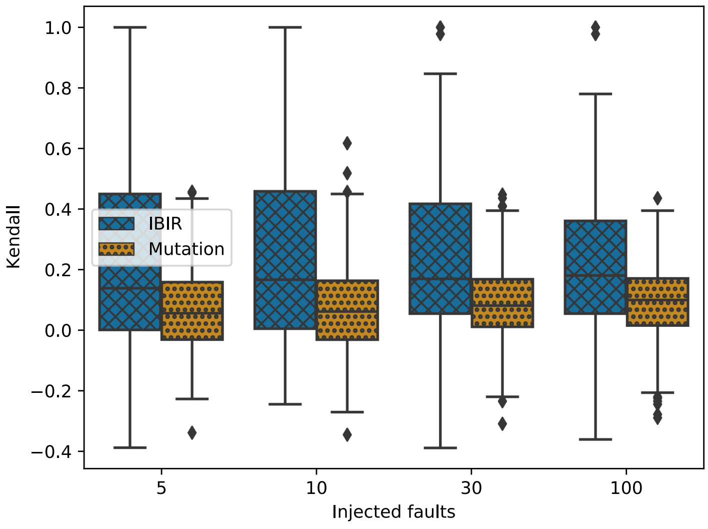
8 DISCUSSION

The effectiveness of iBiR in generating faults that are similar to real ones is endorsed by its two main components: the IRFL and the mutator. The IRFL indicates where the faults need to be injected, and the mutator decides what changes should be made depending on the AST tree of each location.

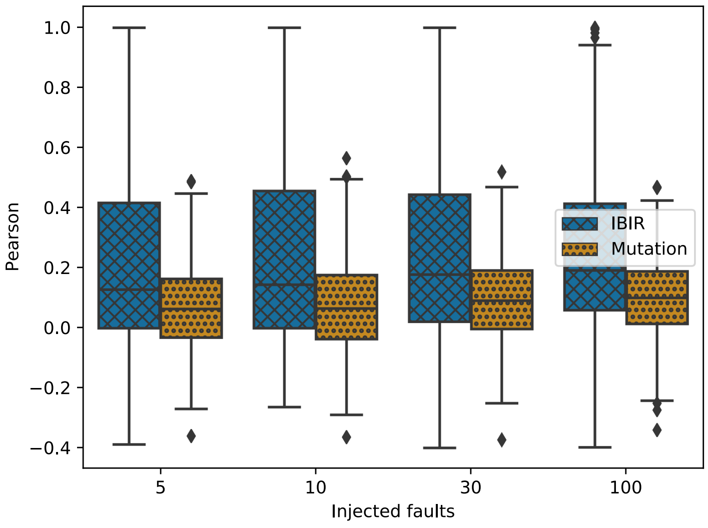
Particularly, compared to conventional mutation testing, we can see that the IRFL is narrowing down the area of injection to the source-code features described by the bug report, while the patterns set of iBiR extends the injection possibilities in that area. On the other hand, conventional mutation testing targets all the source code and injects faults only in statements where their operators are applicable. For instance, applying the typical mutation operators—the `Mutate Operators` and `Remove Statement` ones—on a specific area of code would not induce any fault, if no statement can be removed without breaking the compilation or there is no operator to mutate. In such case, iBiR may inject faults by applying other patterns like mutating the method invocation or the used parameters or inserting a statement.

8.1 Injecting Large Number of Faults

Figure 8 shows that iBiR injects many more faults that couple with the real ones than conventional mutation testing. In fact, it achieves a higher coupling percentage when injecting only 10 faults



(a) Kendall



(b) Pearson

Fig. 9. Correlation coefficients of test suites (samples from the original project test suite). The two related variables are (a) the percentage of injected faults that were detected by the sampled test suites and (b) whether the targeted fault was detected or not by the same test suites.

than the percentage achieved by conventional mutation testing when injecting 1,000 faults. We can see also that when injecting 1,000 faults we achieve the coupling percentages of 61.1% and 18.2% for, respectively, iBiR and mutation testing. This is obviously because the more faults we inject, the more chances we have to inject faults that couple with the real ones. Considering that injecting more faults comes with a considerable consequent cost increase, as the practitioners will

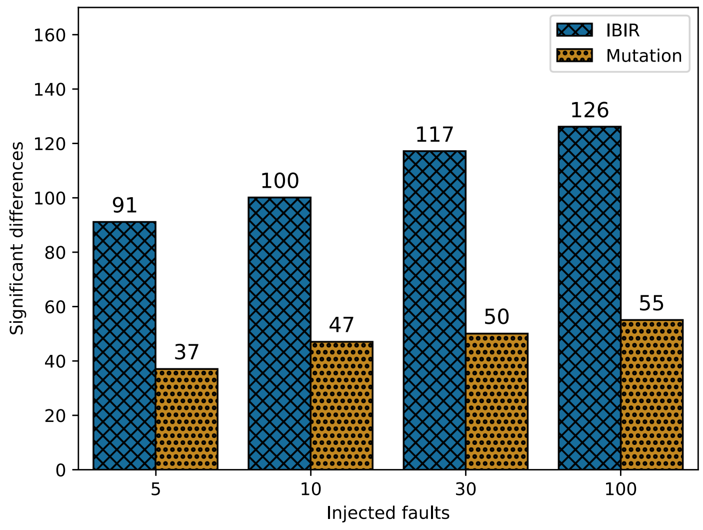


Fig. 10. Number of (real) faults where injected faults provided good indications of fault detection, particularly number of cases with statistically significant difference, in terms of ratios of injected faults detected, between failing and passing test suites (w.r.t. real faults).

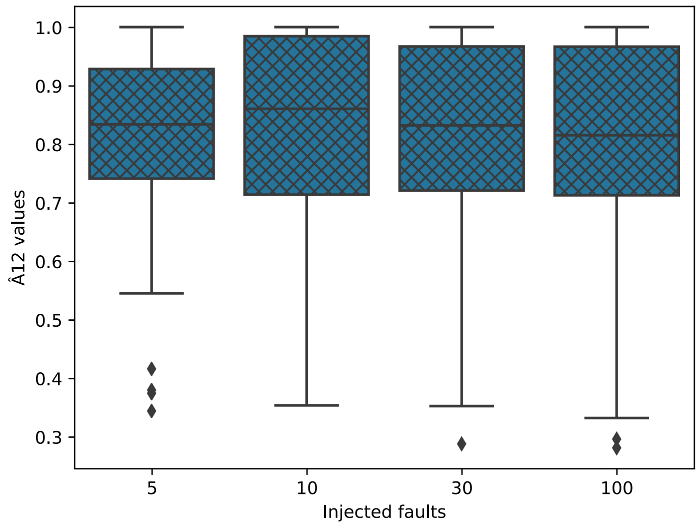


Fig. 11. Vargha and Delaney values for iBiR. $\hat{A}_{12}$ values are computed on the detection ratios of injected faults of the test suites that detect and do not detect the (real) faults.

need more time to analyze the produced mutants, this option is often not favored in practice, where it is better to have few relevant faults than many.

To have a better understanding of the impact of injecting multiple faults, we illustrate in Table 4 the averaged faults coupling success rates when injecting 5, 10, 30, 100, 200, 500, and 1,000 faults with iBiR and mutation testing. We define the success rate as the percentage of coupled faults among all the injected ones. As an example, a coupling success rate of 5% corresponds to 5 coupled faults when injecting 100 faults. In our study, iBiR achieves a much higher success rate

Table 4. Percentage of Injected Faults That Are Coupled to Real Ones When Injecting 5 to 1,000 Faults

Number of Injected Faults	5	10	30	100	200	500	1000
IBIR	5.93%	5.61%	5.78%	5.23%	4.57%	3.43%	2.57%
Mutation	0.29%	0.04%	0.05%	0.06%	0.06%	0.07%	0.07%

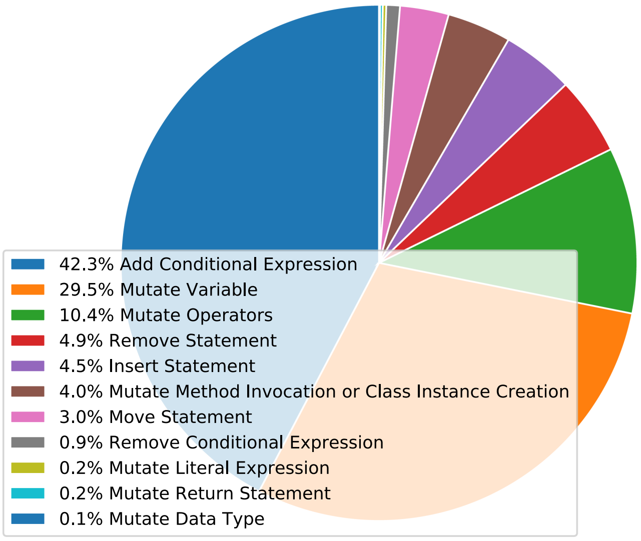


Fig. 12. Distribution of the patterns inducing mutants with an Ochiai coefficient higher than 0.8 for iBiR when injecting 1,000 faults.

than mutation testing: 20, 87, 49, and 36.7 times higher when injecting 5, 100, 500, and 1,000 faults. Even if the coupling percentage increases by injecting more faults (Figure 8), we can see that the more we inject faults, the more the success rate decreases for iBiR. This is a direct consequence of the decrease of the injection-locations likelihood to be related to the targeted bug report. As we explain further in Section 4, iBiR starts by injecting faults in the highly ranked code locations found by its IRFL, then iterates further until all locations are treated or the requested number of faults has been injected. So the higher the requested number of injected faults is, the more faults in lower-ranked locations are injected. On the other hand, we see that the success rate of conventional mutation testing remains relatively low and far behind the one of iBiR. For instance, it remains at 0.07% even when doubling the number of injected faults from 500 to 1,000. In Table 4, we notice that injecting five faults with mutation testing achieves a success rate of 0.29%, which is much higher than the ratios achieved when injecting more faults by the same technique. This is caused by the randomness in the conventional mutation testing results.

8.2 Distribution of the Patterns Inducing Most Effective Injections

To understand better the impact of the used patterns in injecting faults that are similar to real ones, we grouped the faults by their creating patterns and compared the sizes of each group. Figure 12 illustrates the proportion of every pattern's induced faults that have high Ochiai Coefficients (more than 0.8) when injecting 1,000 faults by IBIR in the current dataset. Clearly, more than 70% of the faults with high similarity coefficients have been generated by patterns that are not commonly used in conventional mutation testing techniques: mainly by adding conditional expressions (42.3%) or by mutating variables (29.5%). This is significantly higher than the 15.3% generated with the commonly used conventional mutation operators (10.4% by mutating operators and 4.9% by removing statements). This highlights the fact that iBiR's patterns are bringing a clear advantage over mutation testing.

These percentages and the general performance of every pattern depend on the targeted bug report and the project nature. For instance, the low percentages of multiple patterns in Figure 12 can be the consequence of multiple factors, such as (1) the fact that some faults are occurring less frequently in the current dataset, (2) the fact that some patterns are only applicable on a few specific statement ASTs, or (3) that some patterns produce relatively more mutants in the same location and thus have higher percentages (i.e., the "Mutate Method Invocation," which induced Fault 1 and Fault 2 in the same statement in Figure 3 in Section 4.4).

8.3 iBiR vs. Typical Mutation Operators

Early research on mutation testing defined mutation operators based on all possible simple removals or replacements of programming language elements [3, 27]. This practice was then adopted when defining mutation operators for other languages, such as Java, and in defining object-oriented related mutants [43, 54]. To reduce the number of mutants involved, many tool developers applied a restrictive set of mutation operators, usually referred to as the 5-operator set, based on the selective mutation testing studies performed by Offutt et al. [53, 55] with the result that the majority of modern mutation testing tools implement a version of this 5-operator set together with some deletion operators [34, 60].

In view of the above, all the iBiR injections that involve addition of code elements, i.e., "Insert Statement" and "Mutate Return Staement" categories of Table 2, are fundamentally different from what has been used in mutation testing studies over the years. The "Mutation Literal Expression" category is also something that has not been used by mutation testing studies. The rest of the operators have some similarities with operators used in some studies overall differing significantly from the operators used by any single tool or study. In the following we provide a detailed list of iBiR operators and their related similarities (or novelties) with respect to other studies.

Operators that have not been used by other studies:

- Insert Statement: *Insert a method call, Insert a return statement, Wrap a statement with a try-catch, Insert an if checker.*
- Mutate Conditional Expression: *Insert a conditional expression.*
- Mutate Float or Double Division: *Remove a float or a double cast from the divisor, Remove a float or a double cast from the dividend, Replace float or double multiplication by an int division.*
- Mutate Literal Expression: *Change Boolean, number, or string literals in a statement by another literal or expression of the same type.*
- Mutate Return Statement: *Replace a return expression by an other one.*

Operators that have similarities with those used by other studies:

- Mutate Class Instance Creation: *Replace an instance creation call by a cast of the super.clone() method call.* Similar to the class mutation operators of MuJava [54].
- Mutate Data Type: *Change the declaration type of a variable, Change the casting type of an expression.* Similar to the interface mutation in C [3, 13].
- Mutate Method Invocation: *Replace a method call by another one, Replace a method call argument by another one, Remove a method call argument, Add an argument to a method call.* Similar to the interface mutation [13].

- Mutate Variable: *Replace a variable by another variable or an expression of the same type.* Similar to the variable mutations in C [3].
- Move Statement: *Move a statement to another position.* Similar to the move out of a loop operators in C [3, 13].

Operators that are frequently used by other studies:

- Mutate Conditional Expression: *Remove a conditional expression, Change the conditional operator* [3, 27].
- Remove Statement: *Remove a statement, Remove a method* [5, 27, 34].
- Mutate Operators: *Replace an Arithmetic operator, Replace an Assignment operator, Replace a Relational operator, Replace a Conditional operator, Replace a Bitwise or a Bit Shift operator, Replace a Unary operator, Change arithmetic operations order* [5, 27, 34].

8.4 Project Size and iBiR's Effectiveness

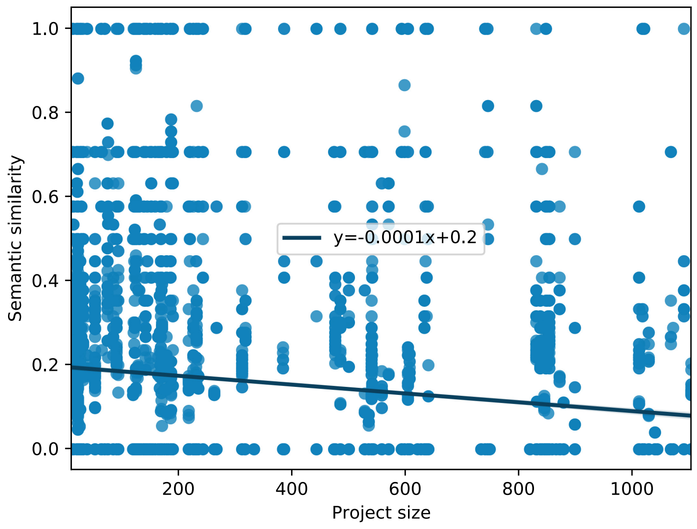
Considering the fault injection as a search task where the target is injecting faults similar to real ones and the search space is the combination of the source code locations and mutation possibilities, we were interested in assessing iBiR's performance for different project sizes. Figures 13(a) and 13(b) show the scatter plots of the semantic similarity by the project size in terms of number of classes. Figures 13(a) and 13(b) consider respectively all the injected faults and the faults having an Ochiai coefficient higher than zero. We can see that the project size has no impact on the effectiveness of iBiR.

9 THREATS TO VALIDITY AND LIMITATIONS

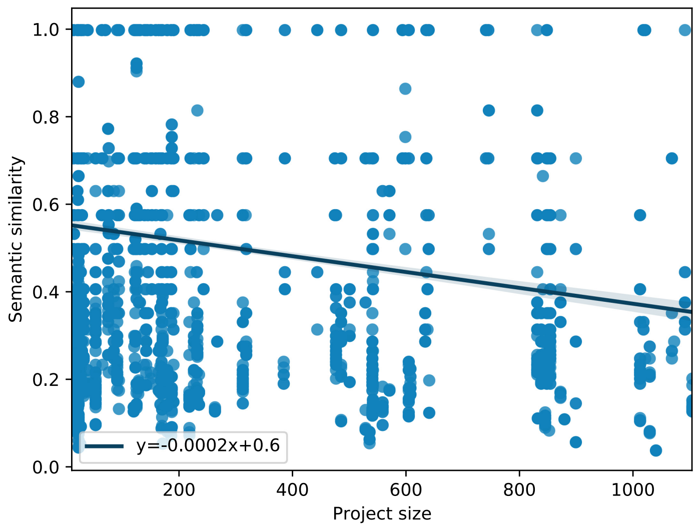
The question of whether our findings generalize forms a typical threat to the validity of empirical studies. To reduce this threat, we used real-world projects, developer test suites, real faults, and their associated bug reports from an established and independently built benchmark. Still, though, we have to acknowledge that these may not be representative of projects from other domains. In addition, as the approach's injection depends on the input bug reports, its effectiveness may be impacted by the content of the reports, such as partial/incomplete or vague descriptions. To reduce this threat, we have run our experiments with all available bug reports in the studied dataset without any particular selection and gotten encouraging results. We acknowledge though that the results may vary depending on the information provided in the reports. In practice, one should make a careful selection of bug reports based on which iBiR could be applied to avoid such cases. Nevertheless, the appropriate selection of bug reports falls outside the scope of this work and has been left open for future research.

Other threats may also arise from the way we handled the injected faults and mutants that were not killed by any test case. We believe that this validation process is sufficient since the test suites are relatively strong and somehow form the current state of practice; i.e., developers tend to use this particular level of testing. In case the approach is put into practice though, things might be different. We also applied our analysis on the fixed program version provided by Defects4J. This was important in order to show that we actually inject the actual targeted faults. Our results might not hold on the cases that the code has drastically changed since the time of the bug report though. We believe that this threat is not of actual importance as we are concerned with fault injection at interesting program locations, which should be pinpointed by the fault localization technique we use. Still, future research should shed some light on how useful these locations and faults are.

Furthermore, some implementation changes of iBiR may improve its usability. For instance, adding an advanced integrity check before applying the patterns would shorten the execution time of the tool. As currently the generated faulty programs are mainly validated via the compilation,



(a) All injected faults.



(b) Faults with an Ochiai coefficient higher than zero.

Fig. 13. Correlation between the semantic similarities and the project size (100 injected faults per targeted (real) fault).

only 52% of the mutants are compilable and thus outputted, while the rest are discarded. Also, one can consider using the same approach with different IRFL techniques. This would eliminate the training cost and reduce the eventual risk of threats that may be induced by the machine learning module currently used to rank the suspicious files. In fact, some of the projects in our evaluation set have been used during the training phase of the latter. Although we did not notice any bias or bad impact on our results, we are aware that this can be considered as an additional threat to validity. However, these threats concern only the file-level localization of the IRFL and not the statement-level one; thus, they would not impact its results. This is because the IRFL is performing a VSM cosine similarity to rank the suspicious statements without involving any machine learning technique in this step, as explained in Section 4.1.

Finally, our evaluation metrics may induce some additional threats. Our comparison basis measurement, i.e., number of injected faults, approximates the execution cost of the techniques and their chances to provide misleading guidance [62], while the fault couplings and semantic similarity metrics approximate the effectiveness of the approaches. These are intuitive metrics, used by previous research [9, 32], and aim at providing a common ground for comparison.

10 RELATED WORK

Software fault injection [73] has been widely studied since the 1970s. Injected faults have been used for the purpose of testing [60], debugging [41, 61], assessing fault tolerance [51], risk analysis [11, 72], and dependability evaluation [7].

Despite the many years of research, the majority of previous research is focused on the fault types. In mutation testing research, mutation operators (fault types) are usually designed based on the grammar of the targeted language [4, 60], which are then refined through empirical analysis, aiming at reducing the redundancy between the injected faults [45, 53]. The most prominent mutant selection approach is that of Offutt et al. [53], which proposed a set of five mutation operators. This set has been incorporated in most of the modern mutation testing tools [28] and is the one that we use in our baseline.

Brown et al. [8] aimed at inferring fault patterns from bug fixes. Their results showed that a large number of mutation operators could be inferred. Along the same lines, Tufano et al. [70] developed a neural machine translation tool that learns to mutate through bug fixes. Key assumptions of these methods are (1) the availability of a comprehensive number of clean bug fixing commits and (2) the absence of fault couplings [52], which are often not met and can often be reduced to what simple mutations do. For instance, the study of Brown et al. found that with few exceptions, almost all mutation operators designed based on the C language grammar appeared in the inferred operator set. Perhaps more importantly, the studies of Natella et al. [51] and Chekam et al. [9] found that the pair of mutant location and type is what makes mutants powerful and not the type itself. Nevertheless, the iBiR goal is complementary to the above studies as it aims at injecting faults that mimic specifically targeted faults, those described in bug reports. This way, one can inject the most important and severe faults experienced.

Some studies attempt to identify the program locations where to inject faults. Sun et al. [68] suggested injecting faults in diverse places within different program execution paths. Gong et al. [21] used graph analysis to inject faults in different and diverse locations of the program spectra. Mirshokraie et al. [48] employed complexity metrics together with actual program executions to inject faults at places with good observability. These strategies aim at reducing the number of injected faults and not to mimic any real fault as our approach. Moreover, their results should be resembled by the random mutant sampling baseline that we use.

Random mutant sampling forms a natural cost reduction method proposed since the early days of mutation testing [14]. Despite that, most of the mutant selection methods fail to perform better than it. Recently, Kurtz et al. [32] and Chekam et al. [9] demonstrated that selective mutation and random mutant sampling perform similarly. From this, it should be clear that despite the advances in selective mutation, the simple random sampling is one of the most effective fault injection techniques. This is the reason we adopt it as a baseline in our experiments. There are also attempts to combine random and selective mutation [83], but they are not relevant for us as they inject numerous mutants.

Natella et al. [51] used complexity metrics as machine learning features and applied them on a set of examples in order to identify (predict) which injected faults have the potential to emulate well the behavior of real ones. Chekam et al. [9] also used machine learning, with many static mutant-related features to select and rank mutants that are likely fault revealing (have high chance to couple with a fault). These studies assume the availability of historical faults and do not aim at injecting specific faults as done by iBiR.

The relationship between injected and real faults has also received some attention [60]. The studies of Papadakis et al. [62], Just et al. [25], and Andrews et al. [6] investigated whether mutant kills and fault detection ratios follow similar trends. The results show the existence of a correlation and, thus, that mutants can be used in controlled experiments as alternatives to real faults. In the context of testing, i.e., using mutants to guide testing, injected faults can help identify corner cases and reveal existing faults. The studies of Frankl et al. [19], Li et al. [36], and Chekam et al. [10] demonstrated that guidance from mutants leads to significantly higher fault revelation than that of other test techniques (test criteria).

11 CONCLUSION

We presented iBiR; a bug-report-driven fault injection tool. iBiR (1) equips researchers with faults (to inject) targeting the critical functionality of the target systems, (2) mimics real faulty behavior, and (3) makes relevant fault injection.

iBiR's use case is simple: given a program and some bug reports, it injects faults emulating the related bugs; i.e., iBiR generates few faults per target bug report. This allows constructing realistic fault pools to be used for test or fault tolerance assessment.

This means that iBiR's faults can be used as substitutes of real faults, in controlled studies. In a sense, iBiR can bring the missing realism into fault injection and therefore support empirical research and controlled experiments. This is important since a large number of empirical studies rely on artificially injected faults [59], the validity of which is always in question.

While the use case of iBiR is in research studies, the use of the tool can have applications in a wide range of software engineering tasks. It can, for instance, be used for asserting that future software releases do not introduce the same (or similar) kind of faults. Such a situation occurs in large software projects [56], where iBiR could help by checking for some of the most severe faults experienced. Testers could also use iBiR for testing all system areas that could lead to similar symptoms as the ones observed and resolved. This will bring benefits when testing software clones [49] and similar functionality implementations.

Another potential application of iBiR is fault tolerance assessment, by injecting faults similar to previously experienced ones and analyzing the system responses and overall dependability. We hope that we will address these points in the near future.

To support this research and enable reproducibility, we have made our data and code available [1].

REFERENCES

[1] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon. 2022. IBIR. Serval, SnT, University of Luxembourg. https://github.com/serval-uni-lu/IBIR.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2009. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. 88–99.

[3] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. 1989. *Design of Mutant Operators for the C Programming Language*. Techreport SERC-TR-41-P. Purdue University, West Lafayette, Indiana.

[4] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press. https://doi.org/10.1017/CBO9780511809163

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624. https://doi.org/10.1109/TSE.2006.83

[6] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.* 32, 8 (2006), 608–624. https://doi.org/10.1109/TSE.2006.83

[7] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. 1993. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Trans. Computers* 42, 8 (1993), 913–923. https://doi.org/10.1109/12.238482

[8] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas W. Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, 511–522. https://doi.org/10.1145/3106237.3106280

[9] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empirical Software Engineering* 25, 1 (2020), 434–487. https://doi.org/10.1007/s10664-019-09778-7

[10] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. 597–608. https://doi.org/10.1109/ICSE.2017.61

[11] Jörgen Christmansson and Ram Chillarege. 1996. Generation of error set that emulates software faults based on field data. In *Digest of Papers: The 26th Annual International Symposium on Fault-Tolerant Computing, 1996 (FTCS-26)*. IEEE Computer Society, 304–313. https://doi.org/10.1109/FTCS.1996.534615

[12] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (Sept. 1990), 391–407.

[13] Márcio Eduardo Delamaro, José Carlos Maldonado, and Aditya P. Mathur. 2001. Interface mutation: An approach for integration testing. *IEEE Trans. Software Eng.* 27, 3 (2001), 228–247. https://doi.org/10.1109/32.910859

[14] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[15] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th SANER*. IEEE, 349–358.

[16] Christiane Fellbaum. 1998. *WordNet: An Electronic Lexical Database*. Bradford Books.

[17] Michael Fischer, Martin Pinzger, and Harald C. Gall. 2003. Populating a release history database from version control and bug tracking systems. In *19th International Conference on Software Maintenance (ICSM'03), The Architecture of Existing Systems, 2003*. IEEE Computer Society, 23. https://doi.org/10.1109/ICSM.2003.1235403

[18] William B. Frakes and Ricardo Baeza-Yates. 1992. *Information Retrieval: Data Structures and Algorithms* (st1 ed.). Prentice Hall.

[19] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. 1997. All-uses vs mutation testing: An experimental comparison of effectiveness. *J. Syst. Softw.* 38, 3 (1997), 235–253. https://doi.org/10.1016/S0164-1212(96)00154-9

[20] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Trans. Software Eng.* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[21] Dunwei Gong, Gongjie Zhang, Xiangjuan Yao, and Fanlin Meng. 2017. Mutant reduction based on dominance relation for weak mutation testing. *Information & Software Technology* 81 (2017), 82–96. https://doi.org/10.1016/j.infsof.2016.05.001

[22] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[23] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 12–23.

[24] 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 (ISSTA'14)*. 437–440.

[25] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014*. 654–665. https://doi.org/10.1145/2635868.2635929

[26] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th ICSE*. IEEE, 802–811.

[27] K. N. King and A. Jefferson Offutt. 1991. A Fortran language system for mutation-based software testing. *Softw. Pract. Exper.* 21, 7 (1991), 685–718.

[28] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empir. Softw. Eng.* 23, 4 (2018), 2426–2463. https://doi.org/10.1007/s10664-017-9582-5

[29] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&C: A divide-and-conquer approach to IR-based bug localization. *arXiv preprint arXiv:1902.02703* (2019).

[30] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 13th Joint Meeting on Foundations of Software Engineering (FSE'19).*

[31] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[32] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16).* 571–582. https://doi.org/10.1145/2950290.2950322

[33] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. 2015. Error models for the representative injection of software defects. In *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT) und Wirtschaftsinformatik (WI) (FA WI-MAW'15) (LNI, Vol. P-239).* GI, 118–119.

[34] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. 2017. Assessing and improving the mutation testing practice of PIT. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'17).* IEEE, 430–435.

[35] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque. 2017. Assessing and improving the mutation testing practice of PIT. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'17).* 430–435. https://doi.org/10.1109/ICST.2017.47

[36] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. 2009. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *2nd International Conference on Software Testing Verification and Validation (ICST,'09), Workshops Proceedings.* IEEE Computer Society, 220–229. https://doi.org/10.1109/ICSTW.2009.30

[37] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05).* 15–26.

[38] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering* 47, 1 (2018), 165–188.

[39] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19).* 1–12.

[40] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19).* 31–42.

[41] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? A unified debugging approach. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20), Virtual Event.* ACM, 75–87. https://doi.org/10.1145/3395363.3397351

[42] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. 2010. Bug localization using latent Dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972–990.

[43] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An automated class mutation system. *Softw. Test. Verification Reliab.* 15, 2 (2005), 97–133. https://doi.org/10.1002/stvr.308

[44] Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing* (1st ed.). MIT Press, Cambridge, Mass.

[45] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. 2018. Time to clean your test objectives. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 456–467. https://doi.org/10.1145/3180155.3180191

[46] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The Cardumen mode of Astor. In *Proceedings of the 10th SSBSE.* Springer, 65–86.

[47] Mark T. Maybury. 2005. Karen Spärck Jones and summarization. In *Charting a New Course: Natural Language Processing and Information Retrieval.* Springer, 99–103.

[48] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Guided mutation testing for javascript web applications. *IEEE Trans. Software Eng.* 41, 5 (2015), 429–444. https://doi.org/10.1109/TSE.2014.2371458

[49] Manishankar Mondal, Md. Saidur Rahman, Ripon K. Saha, Chanchal K. Roy, Jens Krinke, and Kevin A. Schneider. 2011. An empirical study of the impacts of clones in software maintenance. In *The 19th IEEE International Conference on Program Comprehension (ICPC'11)*. IEEE Computer Society, 242–245. https://doi.org/10.1109/ICPC.2011.14

[50] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *7th IEEE International Conference on Software Testing, Verification and Validation (ICST'14)*. IEEE Computer Society, 153–162. https://doi.org/10.1109/ICST.2014.28

[51] Roberto Natella, Domenico Cotroneo, João Durães, and Henrique Madeira. 2013. On fault representativeness of software fault injection. *IEEE Trans. Software Eng.* 39, 1 (2013), 80–96. https://doi.org/10.1109/TSE.2011.124

[52] A. Jefferson Offutt. 1992. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (1992), 5–20. https://doi.org/10.1145/125489.125473

[53] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118. https://doi.org/10.1145/227607.227610

[54] A. Jefferson Offutt, Yu-Seung Ma, and Yong-Rae Kwon. 2006. The class-level mutants of mujava. In *Proceedings of the International Workshop on Automation of Software Test (AST'06)*. 78–84.

[55] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering*. 100–107. http://portal.acm.org/citation.cfm?id=257572.257597

[56] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 305–318.

[57] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. 2018. Mutant quality indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICST Workshops'18)*. IEEE Computer Society, 32–39. https://doi.org/10.1109/ICSTW.2018.00025

[58] Mike Papadakis, Márcio Eduardo Delamaro, and Yves Le Traon. 2014. Mitigating the effects of equivalent mutants with mutant classification strategies. *Sci. Comput. Program.* 95 (2014), 298–319. https://doi.org/10.1016/j.scico.2014.05.012

[59] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. 354–365. https://doi.org/10.1145/2931037.2931040

[60] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter six - Mutation testing advances: An analysis and survey. *Advances in Computers* 112 (2019), 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[61] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5–7 (2015), 605–628.

[62] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 537–548. https://doi.org/10.1145/3180155.3180183

[63] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th ISSTA*. ACM, 199–209.

[64] Gang Qian, Shamik Sural, Yuelong Gu, and Sakti Pramanik. 2004. Similarity between euclidean and cosine angle distance for nearest neighbor queries. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*. Association for Computing Machinery, New York, NY, 1232–1237. https://doi.org/10.1145/967900.968151

[65] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 345–355.

[66] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. 648–659.

[67] Gerard Salton and Michael J. McGill. 1986. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY.

[68] Chang-ai Sun, Feifei Xue, Huai Liu, and Xiangyu Zhang. 2017. A path-aware approach to mutant reduction in mutation testing. *Information & Software Technology* 81 (2017), 65–81. https://doi.org/10.1016/j.infsof.2016.02.006

[69] Stephen W. Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E. Hassan. 2013. The impact of classifier configuration and classifier combination on bug localization. *IEEE Trans. Software Eng.* 39, 10 (2013), 1427–1443. https://doi.org/10.1109/TSE.2013.27

[70] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*. IEEE, 301–312. https://doi.org/10.1109/ICSME.2019.00046

[71] András Vargha and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101

[72] Jeffrey M. Voas, Frank Charron, Gary McGraw, Keith W. Miller, and Michael Friedman. 1997. Predicting how badly "good" software can behave. *IEEE Softw.* 14, 4 (1997), 73–83. https://doi.org/10.1109/52.595959

[73] Jeffrey M. Voas and Gary McGraw. 1997. *Software Fault Injection: Inoculating Programs against Errors*. John Wiley & Sons, Inc..

[74] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. 1–11.

[75] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. 53–63.

[76] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, 1–11. https://doi.org/10.1145/3180155.3180233

[77] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 262–273.

[78] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. 181–190.

[79] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. 2010. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software* 83, 2 (2010), 188–208.

[80] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[81] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. 2015. Bug localization based on code change histories and bug reports. In *Proceedings of the 2015 Asia-Pacific Software Engineering Conference (ICSE'15)*. 190–197.

[82] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2019. Predictive mutation testing. *IEEE Trans. Software Eng.* 45, 9 (2019), 898–918. https://doi.org/10.1109/TSE.2018.2809496

[83] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 92–102. https://doi.org/10.1109/ASE.2013.6693070

[84] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE'12)*. 14–24.

[85] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE'12)*. IEEE, 14–24.

Received 7 April 2021; revised 2 March 2022; accepted 13 May 2022