ReClues: Representing and indexing failures in parallel debugging with program variables

Yi Song
School of Computer Science,
Wuhan University
Wuhan, China
yisong@whu.edu.cn

Xihao Zhang[*]
School of Computer Science,
Wuhan University
Wuhan, China
zhangxihao@whu.edu.cn

Xiaoyuan Xie[†]
School of Computer Science,
Wuhan University
Wuhan, China
xxie@whu.edu.cn

Quanming Liu
School of Computer Science,
Wuhan University
Wuhan, China
liuquanming@whu.edu.cn

Ruizhi Gao
Sonos Inc.
Santa Barbara, USA
youtianzui.nju@gmail.com

Chenliang Xing
School of Computer Science,
Wuhan University
Wuhan, China
xingchenliang@whu.edu.cn

ABSTRACT

Failures with different root causes can greatly disrupt multi-fault localization, therefore, categorizing failures into distinct groups according to the culprit fault is highly important. In such a failure indexing task, the crux lies in the failure proximity, which comprises two points, i.e., how to effectively represent failures (e.g., extract the signature of failures) and how to properly measure the distance between those proxies for failures. Existing research has proposed a variety of failure proximities. The majority of them extract signatures of failures from execution coverage or suspiciousness ranking lists, and accordingly employ the Euclid or the Kendall tau distances, etc. However, such strategies may not properly reflect the essential characteristics of failures, thus resulting in unsatisfactory effectiveness. In this paper, we propose a new failure proximity, namely, the program variable-based failure proximity, and further present a novel failure indexing approach, ReClues. Specifically, ReClues utilizes the run-time values of program variables to represent failures, and designs a set of rules to measure the similarity between them. Experimental results demonstrate the competitiveness of ReClues: it can achieve 44.12% and 27.59% improvements in faults number estimation, as well as 47.56% and 26.27% improvements in clustering effectiveness, compared with the state-of-the-art technique in this field, in simulated and real-world environments, respectively.

CCS CONCEPTS

- **Software and its engineering** → **Software testing and debugging**.

[*] Co-first author.
[†] Corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0217-4/24/04...$15.00
https://doi.org/10.1145/3597503.3639098

KEYWORDS

Failure proximity, Clustering, Failure indexing, Parallel debugging, Program variable

ACM Reference Format:
Yi Song, Xihao Zhang, Xiaoyuan Xie, Quanming Liu, Ruizhi Gao, and Chenliang Xing. 2024. ReClues: Representing and indexing failures in parallel debugging with program variables. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639098

1 INTRODUCTION

Programs in practice usually contain multiple bugs [16, 23, 34, 65], which may reduce the effectiveness of current fault localization techniques [20]. To alleviate this issue [12, 13], researchers proposed to employ ***parallel debugging***[1], where all *failures*[2] are divided into several disjoint groups according to their root causes [14, 26, 54, 60, 86]. This division[3] process aims to achieve two goals, namely, 1) having the number of generated groups equal to the number of faults (i.e., *correct faults number estimation*), and 2) failed test cases in the same group (referred to as fault-focused group) are triggered by the same fault, and vice versa (i.e., *high clustering effectiveness*). As such, each developer can be allocated to a fault-focused group and thus localize the corresponding fault independently and simultaneously.

The effectiveness of the failure division (a.k.a ***failure indexing***) is the core to determine the cost and the performance of parallel debugging. In case of over-division (i.e., the predicted number of faults exceeds the truth), redundant developers will be expropriated for the debugging, resulting in labor waste. And in case of under-division (i.e., the predicted number of faults is less than the truth), more than one iteration of debugging is needed. In fact, prior studies have proven that the higher the accuracy of clustering, the better the performance of parallel debugging, and vice versa [43, 74].

[1] In parallel debugging, people generally consider faults that do not interfere with one another.
[2] Also known as *failed test case* in the context of dynamic testing. We use these two terms interchangeably hereafter.
[3] In the current field of parallel debugging, *clustering* is typically utilized for such *division*. Thus we use these two terms interchangeably hereafter.

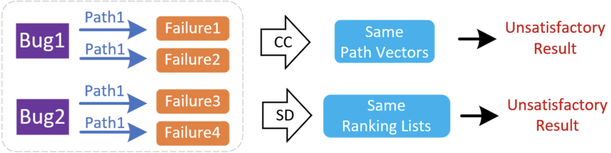


Figure 1: Drawbacks of the SOTA technique

It is well-recognized that there are three essential factors in failure indexing [47]: the fingerprinting function, the distance metric, and the clustering algorithm. The fingerprinting function is responsible for failure representation (e.g., by extracting signatures of failures), which produces the proxy for failed test cases. The distance metric measures the similarity between proxies for failures, which relies on the form of the signature extracted by the fingerprinting function. The clustering algorithm divides failures according to culprit root causes incorporating the distance information. Therefore, the core of failure indexing actually lies in defining a proper fingerprinting function and designing a tailored distance metric (these two components are called ***failure proximity***), and then a proper clustering algorithm can be selected to fit the failure proximity [30].

A large number of researchers have dedicated their effort to exploring this topic [24, 45, 47, 48, 63], yielding a variety of failure proximities. Among them, the code coverage (CC)-based and the statistical debugging (SD)-based failure proximities are the most widespread and commonly-used, previous studies have demonstrated their advancement [9, 67, 85]. The CC-based strategy employs coverage information of binary indicators or execution frequency as its fingerprinting function, and typically uses the Euclid distance as its distance metric. And the SD-based strategy uses the coverage information to further produce a suspiciousness ranking list of program entities as the proxy for a failure by incorporating fault localization techniques, which are generally SBFL (Spectrum-based Fault Localization) ones at the statement granularity [33, 76, 77], and typically employs the Kendall tau distance [39], the Euclid distance, etc. to measure the similarity between the proxies [29, 39]. However, both of the two strategies have drawbacks, as shown in Figure 1. Specifically, when the multiple failures with distinct root causes present to have an identical coverage profile, neither the SD-based nor the CC-based strategy can effectively distinguish the failures, because both of these two strategies essentially rely on program coverage. Unfortunately, such an *identical coverage* phenomenon has been demonstrated to be very common in practice by a prestigious study [47]. Therefore, neither the CC nor the SD strategy is sufficient for serving as an effective failure representation (we use a motivating example to show this in Section 3).

In fact, one of the biggest bottlenecks of using coverage to represent failures can be partly described by the PIE (Propagation, Infection, and Execution) model [64]. The PIE model has demonstrated that a failure can be detected only if the fault[4] infects the program's internal state. However, with the coverage information only, it is very hard to explore the faulty internal state in depth during the program execution, because **being covered is a necessary but not sufficient condition for triggering a failure**. Thus, coverage cannot extract the signature of failures in deep insight. Based on this intuition, we conjecture that program internal dataflows can be a finer-grained failure representation when using coverage gives rise to unsatisfactory effectiveness[5], because variables' values during the program execution are one reasonable candidate to embody program internal states. In this paper, we propose the program variable-based failure proximity, which represents failures by using run-time program variable information (i.e., the run-time values of program variables, **deeper insight into exploring programs' internal state than coverage**), and measures the distance based on the characteristics of such variable information. According to the intuition of the program variable-based failure proximity, we present ReClues, a novel failure indexing approach to **Re**present and **Clu**ster fail**e**d test case**s**. For the fingerprinting function, ReClues first uses an SBFL technique to determine several riskiest program statements as breakpoints, and then collects run-time program variable information at the preset breakpoints during executing a failed test case. For the distance metric, ReClues designs a two-level framework to measure the similarity between a pair of program variable information that serves as proxies for failures.

In the evaluation, we obtain four projects from SIR [17], *flex*, *grep*, *gzip*, and *sed*, followed by injecting mutated faults into clean programs, to generate 600 simulated faulty versions, and also gather 100 real-world faulty versions from five projects in Defects4J [35], *Chart*, *Closure*, *Lang*, *Math*, and *Time*. Experimental results demonstrate that ReClues exceeds the state-of-the-art failure indexing technique significantly, with increases of 44.12% and 27.59% regarding faults number estimation, as well as increases of 47.56% and 26.27% regarding clustering effectiveness, in simulated and real-world environments, respectively.

This paper makes the following contributions:

- **A novel type of failure proximity.** We propose the program variable-based failure proximity, which uses run-time variable information as the signature of failures. As far as we know, this is the first time that program variables serve as proxies for failures in failure indexing.
- **A promising failure indexing approach.** Following the essence of the program variable-based failure proximity, we present ReClues, a novel failure indexing approach comprising a new fingerprinting function and a tailored distance metric.
- **A comprehensive evaluation.** We use a diversity of benchmarks and select convincing metrics for the experiments, revealing the competitiveness of ReClues.
- **A public repository.** We release the experimental data and code at https://github.com/yisongy/ReClues_Repo, to facilitate any intention of replication or reuse.

2 BACKGROUND

2.1 Parallel Debugging

Many studies show that fault localization will be more difficult if multiple faults co-exist in a program [13, 38, 66, 81, 83]. A main reason lies in the phenomenon of the presence of a fault to cause the

[4] In this paper, we follow the practice of general fault localization to consider the non-omission fault.

[5] This conjecture has been verified by the experiments in Section 6.

ineffectiveness of the fault-localization technique to locate another fault [11, 13, 72]. To tackle this challenge, a natural idea is to localize each fault in an independent environment. Thus, researchers and developers often draw on the idea of parallel debugging, i.e., partitioning failures into groups that target a single fault each [11]. And parallelization can also promote the debugging efficiency. As discussed in Section 1, high-quality parallel debugging needs reasonable failure indexing, where the failure proximity, i.e., failure representation and distance measurement, is essential. Here we introduce these two parts of CC and SD-based failure proximities.

2.2 Failure Representation

In software testing and debugging, test cases are typically in the form of program inputs, while failed test cases are those that produce unexpected outputs. Directly available information of a failed test case only contains two parts, the input (i.e, data fed to the program) and the label (i.e., *failed*). It is quite difficult to index failures with only these two resources, since they are poorly distinguishable.

During running a test case, diverse run-time information is generated, which provides failure representation with powerful support to alleviate the mentioned threat. Based on that, many fingerprinting functions have been proposed. For example:

- **Fingerprinting function of the CC-based failure proximity.** The CC-based failure proximity represents a failure as a numerical vector of program coverage. Specifically, it creates a vector with a length equal to the number of program executable entities (which are typically statements), and sets the value of the i^{th} element as 1 or the execution frequency if a failed test case covers the i^{th} statement during the execution, and 0 otherwise [15, 27, 28].
- **Fingerprinting function of the SD-based failure proximity.** The SD-based failure proximity represents a failure as a suspiciousness ranking list of program entities. Specifically, given a failed test case and successful test cases, it employs a fault localization technique (generally an SBFL one) to calculate the risk of program entities being faulty, and produces a ranking list in which all entities are descendingly ordered by their risk values [24, 32, 45].

Though these two failure proximities are recognized as the most widespread and promising strategies to date, the basic resource on which they rely is still code coverage, whose limitation has been mentioned in the references [26, 41, 49] and will be further revealed in Section 3. A more effective fingerprinting function in deeper insight remains lacking.

2.3 Distance Measurement

Defining a reasonable fingerprinting function is the first step for failure proximity. Once failures are translated into the corresponding proxies, properly measuring the distance among the proxies is of great importance. Designing such a distance metric is not an independent process, since it must match the characteristics of the proxies. For example:

- **Distance metric of the CC-based Failure proximity.** The CC-based failure proximity typically utilizes the Euclid distance, etc., since it represents a failure as a program code coverage vector, and the Euclid distance is a simple way to measure the distance between such numerical vectors.
- **Distance metric of the SD-based Failure proximity.** The SD-based failure proximity typically utilizes the Kendall tau distance [39], the Jaccard distance, the Euclid distance, etc., since it represents a failure as a ranking list, which can be suitably handled by these distance metrics.

It can be seen that the design of the distance metric is highly dependent on the fingerprinting function. To put it another way, a work defining a novel type of fingerprinting function should also design a tailored distance metric at the same time.

3 MOTIVATION

Listing 1: An example program

```
1   public static String process(String s){
2       if(s.contains("*1*") || s.contains("*2*")){
3           return "";}
4       int sign = 0;
5       int sum_1 = 0;
6       sum_1 = s.contains("wordNone") ? 1 : 0;
7       sign += sum_1;
8       s = s.replaceAll("wordNone", "?1?");
            // Fault1: "?1?" should be "*1*"
9       int sum_2 = 0;
10      sum_2 = s.contains("wordNtwo") ? 2 : 0;
11      sign += sum_2;
12      s = s.replaceAll("wordNtwo", "*2*");
13      if(sign == 3){
14          return "both pattern recognized";}
15      String msg = sign == 1 ? "wordNone recognized" :
            "pass";
16      msg = sign > 2 ? "wordNtwo recognized" : msg;
            // Fault2: "> 2" should be "== 2"
17      return s + "//" + msg;}
```

In Figure 1 in Section 1, we describe that when the multiple failures with distinct root causes present to have an identical coverage profile, neither the CC-based nor the SD-based failure proximity can work well. Here we use a motivating example to illustrate such a scenario. The toy program in Listing 1 aims to replace certain words from the input string, and output the modified string and the log message. Specifically, if an input string contains "*wordNone*" or "*wordNtwo*", these two words will be replaced with "**1**" and "**2**", respectively. The log message records the operation of the program. Given a test suite containing 12 test cases: t_1 = "*speak wordNone*", t_2 = "*wordNone*", t_3 = "*wordNonecontained*", t_4 = "*wwwwordNoneeee*", t_5 = "*has wordNtwo*", t_6 = "*wordNtwo*", t_7 = "", t_8 = "*midd*1*le*", t_9 = "**1*2**", t_{10} = "*a normal sentence*", t_{11} = "*wordnonewordNtw*", and t_{12} = "*wordNone and wordNtwo*". Six of them are labeled as $failed$ due to the unexpected outputs: t_1, t_2, t_3 and t_4 are triggered by $Fault_1$, t_5 and t_6 are triggered by $Fault_2$ (we refer to the failed test cases $t_1 \sim t_6$ as $f_1 \sim f_6$, respectively). An ideal failure indexing process should satisfy two goals. The first is to correctly predict the number of clusters (i.e., the number of faults, two). And the second is to properly index failures according to their root cause, i.e., delivering two clusters, $\{f_1, f_2, f_3, f_4\}$ and $\{f_5, f_6\}$.

We can find that all of these six failed test cases have an identical coverage profile, i.e., $\{s_1, s_2, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{15}, s_{16}, s_{17}\}$. Therefore, the CC-based failure proximity will

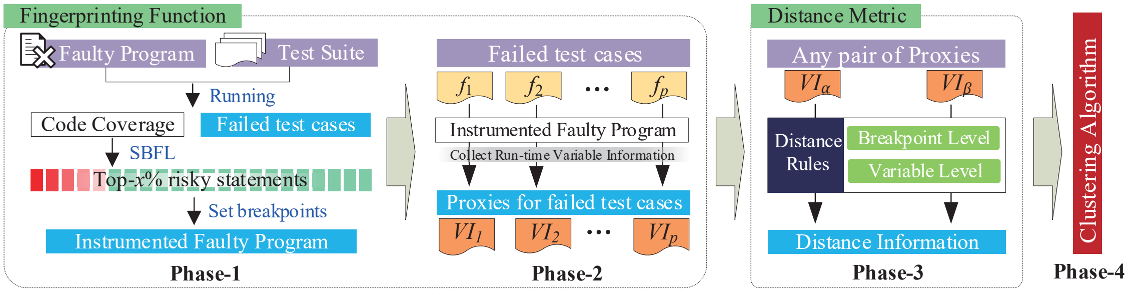


Figure 2: The overview of ReClues

represent these failures identically and thus have no way to distinguish them. Notice that if the gathered coverage information is the same, the SD-based failure proximity will also be trapped, because the incorporation of SBFL techniques cannot handle the scenario where multiple faults are triggered in the same path. That is to say, **even the most widespread and state-of-the-art failure proximities are still not enough to deliver promising outcomes in such an *identical coverage* situation**, which, unfortunately, is very common in practice, as concluded by a well-known study in the field of failure indexing:

> *"A significant portion of execution profiles would be the same even if these failures are due to different faults"* [47].

Thus, developing a better failure proximity is of great significance. To this aim, two questions must be answered: 1) How to define a fingerprinting function to better extract the signature of failures? And 2) How to design a tailored distance metric for measuring the similarity between the proxies for failures?

4 APPROACH

4.1 Overview

The program variable-based failure proximity uses the run-time values of program variables to represent failures, and measures the similarity between failures based on the characteristics of variable information. Following this description, we propose a novel failure indexing approach, ReClues. We depict its overall workflow in Figure 2 and summarize the four phases as follows:

- **Phase-1:** Executing the test suite against the faulty program, inputting the code coverage into a spectrum-based fault localization technique to calculate suspiciousness for each program statement, and determining the Top-x% riskiest statements as breakpoints.
- **Phase-2:** For each failed test case f_i ($i = 1, 2, ..., p$), collecting the run-time variable information during its execution at the preset breakpoints. The variable information will serve as the proxy for the corresponding failure, denoted as VI_i ($i = 1, 2, ..., p$).
- **Phase-3:** Calculating the distance between each pair of VIs using the proposed distance metric.
- **Phase-4:** Delivering the distance information to the downstream clustering algorithm, enabling all failures to be indexed according to the root cause.

Phase-1 and Phase-2 are the fingerprinting function, Phase-3 depicts the distance metric, and Phase-4 is the clustering algorithm. Next we elaborate on the technical details of each.

4.2 Fingerprinting Function

Instead of solely considering code coverage, we dig deeper into available information during the dynamic execution, and use run-time program variable information, i.e., the run-time values of program variables queried at a set of breakpoints during running a failed test case, to better represent this failure. It is obvious that such a fingerprinting function involves two factors: the breakpoint determination and the variable information collection.

4.2.1 Breakpoint determination (Phase-1). We first employ an SBFL technique to calculate risk values for each program statement, then determine whether a statement should be set as a breakpoint (the Top-x% riskiest statements are selected). The intuition we adopt this strategy is that statements with higher risk values are more likely to be faulty, and variable information collected at these positions could have stronger capability to reveal the faults, thus can contribute more to representing failures. We further investigate the impact of the value of x on the effectiveness of ReClues in Section 6.1. As a reminder, our method does not require SBFL to pinpoint the precise faulty location. We only desire a set of highly-risky statements, which are sufficient to identify potential breakpoints for monitoring the execution of faulty programs.

4.2.2 Variable information collection (Phase-2). In ReClues, the proxy for a failure is defined as a two-dimension dictionary. Specifically, the i^{th} failed test case f_i in the test suite can be represented as $VI_i = \{bp_1\text{: } V_1^i, bp_2\text{: } V_2^i, ..., bp_j\text{: } V_j^i, ..., bp_q\text{: } V_q^i\}$, where V_j^i denotes the variable information queried at the j^{th} breakpoints (denoted as bp_j) during the execution of f_i, q is the total number of preset breakpoints. V_j^i is also a dictionary, which contains the name (dictionary's key) and value (dictionary's value) of the queried variables.

When a program stops at a statement, only the variable information at the position before the execution of that statement can be collected. Thus, to make sure that variable information is collected regarding the positions determined by Phase-1, when faulty programs stop at each breakpoint, we continue executing a further step and then carry out the collection operation. In addition, if a statement is covered for more than once, the variables' value is collected when the execution is completed, since the latest value reflects the entire accumulation of the execution. As a reminder, for the sake of cost saving, if there are function calls in a statement, we do not iteratively navigate to the callee to query variable information, but focus on the original location of preset breakpoints (Experiments in Section 6 demonstrate that this strategy is good enough for failure indexing).

4.3 Distance Metric

In failure indexing, a basic concept is that *failures triggered by the same fault should be as similar as possible, and vice versa.* Thus, in the context of ReClues, the intuition of designing a distance metric can be concretized as *the run-time variable information of the failures caused by the same fault should be as similar as possible, and vice versa.* For making ReClues better competent to such a mission, its distance metric is designed with two levels: the breakpoint level and the variable level. Given a pair of failures that requires measuring the distance, f_α and f_β, the breakpoint level divides all of the breakpoints into three categories, i.e, those covered by both f_α and f_β, those covered by only one of f_α and f_β, and those covered by neither of them. Then, the variable level further compares the variable information at the breakpoints fallen into the first category. The detailed descriptions of the two levels are as follows.

4.3.1 *Breakpoint level (Phase-3).*

Given a pair of failed test cases, f_α and f_β, the breakpoint level calculates the distance between them using Formula 1,

$$Distance_{f_\alpha,\, f_\beta} = \frac{\sum_j^q Distance_{bp_j}}{\sum_j^q BPCount_j} \tag{1}$$

where q is the number of preset breakpoints, $Distance_{bp_j}$ is the distance between f_α and f_β at the j^{th} breakpoint bp_j, and $BPCount_j$ is a binary constant for getting the mean of distances[6].

For $Distance_{bp_j}$, we calculate its value according to both Formula 2 and Formula 3.

$$Distance_{bp_j} = \begin{cases} Distance_{var}^j & e_j^\alpha + e_j^\beta = 2 \\ 1 & e_j^\alpha + e_j^\beta = 1 \\ 0 & e_j^\alpha + e_j^\beta = 0 \end{cases} \tag{2}$$

$$e_j^{\alpha|\beta} = \begin{cases} 1 & \text{if } f_\alpha \mid f_\beta \text{ covers } bp_j \\ 0 & \text{if } f_\alpha \mid f_\beta \text{ not covers } bp_j \end{cases} \tag{3}$$

And for $BPCount_j$, we calculate its value according to both Formula 4 and Formula 3.

$$BPCount_j = \begin{cases} 1 & e_j^\alpha + e_j^\beta = 2 \\ 1 & e_j^\alpha + e_j^\beta = 1 \\ 0 & e_j^\alpha + e_j^\beta = 0 \end{cases} \tag{4}$$

It can be seen that all breakpoints are divided into three categories, and as a consequence of which, the calculation of $Distance_{f_\alpha,\, f_\beta}$ also involves three scenarios:

- **For those breakpoints covered by both of the failures** ($e_j^\alpha + e_j^\beta = 2$). $Distance_{bp_j}$ is set to $Distance_{var}^j$, which will be further calculated using the variable information, and such a process is discussed at the variable level in Section 4.3.2. And $BPCount_j$ is set to 1.
- **For those breakpoints covered by only one of the failures ($e_j^\alpha + e_j^\beta = 1$).** It means that two failures have distinct execution paths at those breakpoints, their variable information at those breakpoints should also be regarded as distinct. Thus, $Distance_{bp_j}$ is directly set to the maximum value, 1 (all the values of $Distance_{bp_j}$ will be normalized to the interval of [0, 1]). And $BPCount_j$ is set to 1.
- **For those breakpoints covered by neither of the failures ($e_j^\alpha + e_j^\beta = 0$).** We think it is difficult for them to make any contribution to failure indexing. Therefore, $Distance_{bp_j}$ is directly set to 0, and $BPCount_j$ is also set to 0, to make sure they have no impact on the outcome of $Distance_{f_\alpha,\, f_\beta}$.

4.3.2 *Variable level (Phase-3).*

As defined in Formula 2, if failed test cases f_α and f_β both cover bp_j, $Distance_{bp_j}$ will be set to $Distance_{var}^j$, which is with in the scope of the variable level. We calculate the value of $Distance_{var}^j$ using Formula 5,

$$Distance_{var}^j = \begin{cases} \frac{\sum_z^{|\hat{V}_j^\alpha \cup \hat{V}_j^\beta|} dis_z}{|\hat{V}_j^\alpha \cup \hat{V}_j^\beta|} & \text{if } |\hat{V}_j^\alpha \cup \hat{V}_j^\beta| > 0 \\ 1 & \text{if } |\hat{V}_j^\alpha \cup \hat{V}_j^\beta| == 0 \end{cases} \tag{5}$$

where $\hat{V}_j^{\alpha|\beta}$ is the set of all variables' names collected at bp_j while executing f_α or f_β. $\hat{V}_j^\alpha \cup \hat{V}_j^\beta$ stands for the union of all variables' names collected at bp_j while executing f_α and f_β, and $|\hat{V}_j^\alpha \cup \hat{V}_j^\beta|$ is the scale of this union. If the value of $\left|\hat{V}_j^a \cup \hat{V}_j^b\right|$ equals 0, meaning that no variable is collected at bp_j while executing the two failures, we simply set $Distance_{var}^j$ to the maximum value (i.e., 1) to handle this situation. Otherwise, We use a *variable-to-variable* tactic to measure the similarity between f_α and f_β at bp_j. Specifically, each of the variables in $\hat{V}_j^\alpha \cup \hat{V}_j^\beta$ will be compared with its counterpart (i.e., the same variable collected during executing the other failure).

Such a *variable-to-variable* tactic is implemented by dis_z in Formula 5, which is the distance between the values of the z^{th} variable in the execution of f_α and f_β, as defined in Formula 6, Formula 7, and Formula 8 (For convenience, we denote the z^{th} variable as var_z, and denote the values of var_z during the execution of f_α and f_β as val_z^α and val_z^β, respectively).

$$dis_z = \begin{cases} \text{Jacc}\left(val_z^\alpha,\ val_z^\beta\right) & c_z^\alpha + c_z^\beta = 2 \textbf{ and } n_z^\alpha + n_z^\beta = 0 \\ 1 & c_z^\alpha + c_z^\beta = 2 \textbf{ and } n_z^\alpha + n_z^\beta = 1 \\ 0 & c_z^\alpha + c_z^\beta = 2 \textbf{ and } n_z^\alpha + n_z^\beta = 2 \\ 1 & c_z^\alpha + c_z^\beta = 1 \end{cases} \tag{6}$$

$$c_z^{\alpha|\beta} = \begin{cases} 1 & \text{if } f_\alpha \mid f_\beta \text{ collects } var_z \\ 0 & \text{if } f_\alpha \mid f_\beta \text{ not collects } var_z \end{cases} \tag{7}$$

[6] There could be an uncommon situation that the value of $\sum_j^q BPCount_j$ is zero, which means that two failures do not cover any breakpoint. Setting more breakpoints can handle it, and we investigate this question in Section 6.1.

$$n_z^{\alpha|\beta} = \begin{cases} 0 & \text{if } val_z^{\alpha} \mid val_z^{\beta} \text{ is not null} \\ 1 & \text{if } val_z^{\alpha} \mid val_z^{\beta} \text{ is null} \end{cases} \tag{8}$$

It can be seen that all of the variables are divided into four categories, and as a consequence of which, the calculation of dis_z also involves four scenarios:

- **If a variable is collected by both of the failures, and neither of their values is null** ($c_z^{\alpha} + c_z^{\beta}$ = 2 and $n_z^{\alpha} + n_z^{\beta}$ = 0), we use the Jaccard distance, which is defined in Formula 9, to calculate dis_z.
- **If a variable is collected by both of the failures, and one of the values is null**, ($c_z^{\alpha} + c_z^{\beta}$ = 2 and $n_z^{\alpha} + n_z^{\beta}$ = 1), we assign the maximum value (i.e., 1) to dis_z, since the two values are uncomparable, which shows the divergence between the two failures when it comes to this variable at bp_j.
- **If a variable is collected by both of the failures, and both of the two values are null** ($c_z^{\alpha} + c_z^{\beta}$ = 2 and $n_z^{\alpha} + n_z^{\beta}$ = 2), we assign the minimum value (i.e., 0) to dis_z, since such a condition can be considered as val_z^{α} and val_z^{β} having no difference.
- **If a variable is collected by only one of the failures** ($c_z^{\alpha} + c_z^{\beta}$ = 1), we assign the maximum value (i.e., 1) to dis_z, because this condition indicates that the two failures may have different dataflows.

As mentioned previously, if var_z falls into the first category, the value of dis_z is calculated by the Jaccard distance using Formula 9. This is because in our experiments, we find that regarding a variable's value as a string is beneficial for distance measurement[7], and the Jaccard distance is commonly used in measuring the distance between two strings (it regards a string as a set of multiple characters[8]) [42, 51, 55].

$$\text{Jacc}\left(val_z^{\alpha},\ val_z^{\beta}\right) = \text{norm}\left(1 \text{ - } \frac{|val_z^{\alpha} \cap val_z^{\beta}|}{|val_z^{\alpha} \cup val_z^{\beta}|}\right) \tag{9}$$

The function *norm* is used to achieve the 0-1 normalization, which is employed to make the same scale for the distances in the four scenarios, and is defined in Formula 10.

$$\text{norm}(J_z) = \frac{J_z - \min(J)}{\max(J) - \min(J)} \tag{10}$$

Where J_z is the value of $Jacc\left(val_z^{\alpha},\ val_z^{\beta}\right)$ without the normalization. And $max(J)$ and $min(J)$ are the maximum and the minimum values among all J_z, respectively.

To summarize, Figure 3 depicts the workflow of the distance metric, clarifying the relationship among all of the mentioned formulas: Formula 1 is the core of the distance metric, while Formula 2 and Formula 3 are for its numerator, and Formula 4 and Formula 3 are for its denominator. Formula 5 is called by Formula 2, which can be supported by Formula 6, Formula 7, and Formula 8. And Formula 6 is determined by Formula 9 and Formula 10.

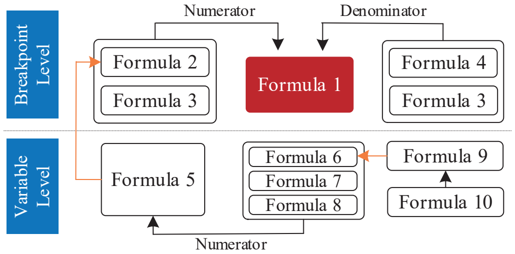


Figure 3: The workflow of the distance metric

In the distance metric, we borrow the idea of SBFL to ignore the execution order of breakpoints. Specifically, SBFL builds linkages between tests and programs through coverage without considering the concrete control flow, and such a strategy for modeling raw data has been widely recognized by academia [76, 78, 84].

4.4 Clustering Algorithm

We employ the clustering component of MSeer [24], the state-of-the-art failure indexing technique, to complete the clustering stage of ReClues (i.e., Phase-4 in Figure 2). This algorithm involves the faults number estimation and the clustering. Next we give a concise description and more details can be found in the reference [24].

4.4.1 The faults number estimation (Phase-4). It is well-recognized that one of the trickiest challenges in clustering lies in the estimation of the number of clusters [22, 40, 62]. Putting it into the context of failure indexing, we can claim that predicting the number of faults given a set of failures is of great importance. The adopted algorithm presents a novel mountain method-based technique inspired by previous works [10, 82], to carry out the faults number estimation and the assignment of initial medoids to clusters simultaneously. Specifically, it first calculates a potential value for each data point (i.e., a failure) according to the density of its surroundings, such a potential value is used to measure the possibility of a failure being set as a medoid. And then, 1) Choosing the failure with the highest potential value as a medoid. 2) Updating the potential values of all failures in accordance with their distance from the newest medoid. 3) Repeating these two processes iteratively, until the maximum potential value falls within a certain threshold.

4.4.2 The clustering (Phase-4). The adopted algorithm utilizes a widely-used clustering technique, K-medoids. The K-medoids technique sets actual (rather than virtual) data points as medoids thus can be more applicable to ReClues, because the mean of variable information is difficult to define. Furthermore, the K-medoids technique has shown to be very robust to the existence of noise or outliers [19, 37].

4.5 Running Example

We recall the toy program in Listing 1 to exemplify the workflow of ReClues step-by-step. Here we give a simplified version of this running example due to the space limitation, the complete version of this example can be found in our public repository[9].

[7] For some special types of variables, further action will be taken. For example, for a pointer, we will further index its value by address.

[8] For instance, for two strings "*abac*" and "*abcd*", the Jaccard distance between them can be calculated as 3 / 4 = 0.75 (the length of the intersection of the sets {'*a*', '*b*', '*c*'} and {'*a*', '*b*', '*c*', '*d*'} is 3, and the length of the union is 4).

[9] https://github.com/yisongy/ReClues_Repo/blob/main/detailedExample.pdf

Table 1: Distance information of the running example

	f_1	f_2	f_3	f_4	f_5	f_6
f_1	0	0.2	0.2	0.2	0.8	1
f_2	0.2	0	0.2	0.2	1	1
f_3	0.2	0.2	0	0.2	0.8	1
f_4	0.2	0.2	0.2	0	1	1
f_5	0.8	1	0.8	1	0	0.2
f_6	1	1	1	1	0.2	0

Table 2: Benchmarks

Language	Project	Version	kLOC	Functionality
C	flex	2.5.3	14.5	Parser generator
	grep	2.4	13.5	Text matcher
	gzip	1.2.2	7.3	File archiver
	sed	3.02	10.2	Stream editor
Java	Chart	2.0.0	96.3	Chart library
	Closure	2.0.0	90.2	Closure compiler
	Lang	2.0.0	22.1	Apache commons-lang
	Math	2.0.0	85.5	Apache commons-math
	Time	2.0.0	28.4	Date and time library

For Phase-1 (Breakpoint determination). Employing an SBFL technique (e.g., DStar [70]) to calculate suspiciousness for each statement, and determine several riskiest statements as breakpoints (the determination threshold will be investigated in Section 6.1, here we take Top-10% as an example). Thus, s_{15} and s_{16} are selected as breakpoints[10].

For Phase-2 (Variable information collection). Collecting the run-time variable information at s_{15} (bp_1) and s_{16} (bp_2) during the execution of the six failed test cases, i.e., representing $f_1 \sim f_6$ as $VI_1 \sim VI_6$, respectively.

For Phase-3 (Distance measurement). The distances between each pair of failures (i.e., each pair of variable information) are calculated and given in Table 1. Revisiting the mapping relationship between the two faults and the six failures, i.e., $Fault_1 : \{f_1, f_2, f_3, f_4\}$ and $Fault_2 : \{f_5, f_6\}$. It can be seen that the failures triggered by the same fault are highly similar, while on the contrary, the failures triggered by different faults show low similarity to each other.

For Phase-4 (Clustering). Running the clustering algorithm, two groups will be produced: $\{f_1, f_2, f_3, f_4\}$ and $\{f_5, f_6\}$. Revisiting the oracle of this example given in Section 3, the clustering result can be found to achieve promising failure indexing, because ReClues properly divides all of the six failures according to the two underlying root causes.

From this example we can observe that the mechanism of ReClues lies in the dataflow. Specifically, when the coverage of the failures having distinct root causes is identical, the run-time values of program variables could play a role of distinguisher. It should be pointed out that although dynamic slices can reflect the dataflow to an extent and thus can preliminarily divide those failures, we are interested in whether there is a finer-grained and more precise representer that can be competent to such a mission.

5 EXPERIMENTAL SETUP

In this section, we introduce the experimental setup of this study, including research questions, parameter setting, datasets, metrics, and environments.

5.1 Research Questions

- **RQ1: The value of the hyperparameter.** In the fingerprinting function, ReClues determines the Top-x% riskiest statements as breakpoints. We investigate how the value of x impacts the effectiveness of ReClues.
- **RQ2: Impact analyses of components.** In the distance metric, ReClues measures the similarity between two sets of variable information through the breakpoint level and the variable level. How does each of the two components impact the effectiveness of ReClues?
- **RQ3: Competitiveness of ReClues.** How does ReClues perform compared with the current most prevalent and promising failure indexing techniques?

5.2 Parameter Setting

As stated in Section 4.2.1, we need to first determine an SBFL technique to calculate risk values for program statements. In the experiment, we use DStar[11], one of the best SBFL techniques [70]. Such a choice is not hard-coded but can be configurable, ReClues can adapt to any other fault localization techniques that are able to deliver a suspiciousness ranking list of program entities at the statement granularity.

5.3 Datasets

5.3.1 Simulated Scenarios. SIR (Software-artifact Infrastructure Repository) is a classical platform for experiments in software testing and debugging [17]. We obtain four C projects from SIR: *flex*, *grep*, *gzip*, and *sed*, which have been extensively adopted in earlier works [8, 25, 59]. Based on these programs, we create 1-bug, 2-bug, 3-bug, 4-bug, and 5-bug faulty versions (i.e., faulty programs containing one, two, three, four, and five bugs, respectively) by employing mutation strategies [50], in light of the fact that previous research has confirmed that mutation-based faults can provide credible results for experiments in software testing and debugging [4, 5, 18, 36, 44, 57]. To create an r-bug faulty version (r = 1, 2, 3, 4, 5), we inject 1, 2, 3, 4, and 5 mutant(s) into the clean program, respectively. This approach (creating multiple-bug faulty versions by injecting bugs from multiple single-bug mutants) has been used by the majority of the published studies [1, 27, 28, 41, 85]. We employ an existing tool with 13 "fork" and 23 "star" on GitHub to perform mutation [7]. It defines 67 types of points that can be mutated, and provides several mutation operators for each one. For example, replacing operators such as addition, subtraction, multiplication, division, etc. with each other [31], and reversing an *if-else* predicate, deleting an *else* statement, modifying a decision condition [80], and so on. The description of the four projects is given in Table 2. In total, we create 600 SIR faulty versions.

[10] If several statements share the same value of suspiciousness, we adopt the widely-used solution by ranking them in the ascending order of line numbers [24, 52, 61, 79].

[11] Considering the preference for DStar in many other studies (such as [6, 53, 69]), we set the value of * in DStar to 2, the most thoroughly-explored value, in our experiments.

Table 3: Scenarios in two types of metrics

Metric	Notation	Results of failure indexing	
		In generated cluster	In oracle cluster
FMI and JC	SS	Same	Same
	SD	Same	Difference
	DS	Difference	Same
	DD	Difference	Difference
PR and RR	TP	Positive	Positive
	FP	Positive	Negative
	TN	Negative	Negative
	FN	Negative	Positive

5.3.2 Real-world Scenarios. Defects4J is one of the most popular benchmarks in the current field of software testing and debugging, due to its realism and ease-to-use [35]. We obtain five Java projects from Defects4J: *Chart*, *Closure*, *Lang*, *Math*, and *Time*, and then based on which search for 1-bug, 2-bug, 3-bug, 4-bug, and 5-bug faulty versions, according to the search strategy proposed by An et al. [3]. Specifically, Defects4J is typically for single-fault scenarios, namely, no matter how many bugs are contained in a faulty program, the provided test suite is only sufficient to reveal one of them. Such a characteristic hinders its use in failure indexing. To adapt Defects4J to multi-fault scenarios, An et al. presented a search strategy to enhance the test suite, that is, transplanting the fault-revealing test case(s) of another faulty version (or other faulty versions) to the original faulty version, allowing a larger-scale test suite to find more faults in the original program. The description of the five projects is given in Table 2. In total, we get 100 Defects4J faulty versions.

5.4 Metrics

Generally speaking, the capability of a failure indexing approach can be measured from two aspects. One is the faults number estimation, namely, to what extent the number of faults can be correctly predicted given a series of observed failures. And the other is the clustering effectiveness, namely, to what extent these failures can be indexed according to the root cause.

5.4.1 Faults number estimation. For an r-bug faulty version, we utilize a failure indexing approach T to estimate the number of faults r. If the estimated number of faults k is equal to r, we mark this faulty version as *equal*, and use V_{equal}^{T} to denote the number of faulty versions that fall into the *equal* category when using T. Obviously, a larger value of V_{equal}^{T} indicates a stronger capability to represent failures of T.

5.4.2 Clustering effectiveness. We employ the Fowlkes and Mallows Index (FMI), the Jaccard Coefficient (JC), the Precision Rate (PR), and the Recall Rate (RR), to measure the effectiveness of a clustering process. These four metrics are classic and ease-to-implement, and they have also been adopted in a collection of prior research [73, 75, 87].

Among them, FMI and JC compare the indexing consistency of each pair of failed test cases in the generated cluster with that in the oracle cluster [21], as shown in Formula 11 and Formula 12. The four possible scenarios in the comparison can be found in Table 3.

$$FMI = \sqrt{\frac{X_{SS}}{X_{SS}+X_{SD}} \times \frac{X_{SS}}{X_{SS}+X_{DS}}} \tag{11}$$

$$JC = \frac{X_{SS}}{X_{SS}+X_{SD}+X_{DS}} \tag{12}$$

Where X_{SS} is the number of pairs of "SS", and so forth.

PR and RR compare the classification result of failed test cases in the generated cluster with that in the oracle cluster, as shown in Formula 13 and Formula 14. The four possible scenarios in the comparison can be found in Table 3.

$$PR = \frac{X_{TP}}{X_{TP}+X_{FP}} \tag{13}$$

$$RR = \frac{X_{TP}}{X_{TP}+X_{FN}} \tag{14}$$

Where X_{TP} is the number of failures of "TP", and so forth.

We deliver only the faulty versions whose number of faults is correctly predicted (hereafter, simply referred to as "$k == r$" faulty versions) to the following clustering phase. The reason behind such a strategy is that, if the predicted number of faults k is not equal to r, it is hard to compare the k generated clusters with the r oracle groups. As a consequence of which, the measurement of clustering effectiveness can be difficult. We use Formula 15 to calculate the sum of the metric values on "$k == r$" faulty versions,

$$S\_M_{M}^{T} = \sum_{i}^{V_{equal}^{T}} M_i \tag{15}$$

where S_M is the abbreviation for "*Sum_Metrics*". V_{equal}^{T} is the number of "$k == r$" faulty versions when using T. M_i is the value of the clustering metric M (M takes FMI, JC, PR, or RR) on the i^{th} "$k == r$" faulty version.

Notice that "$k == r$" is just an ideal scenario (not necessary) for ReClues. Even if $k \neq r$, ReClues can also work, as introduced in Section 1 (the part of "over-division" and "under-division").

5.5 Environments

We collect program coverage and run-time variable information on Ubuntu 16.04.1 LTS with GCC 5.4.0 and JDB 1.8. The distance measurement and clustering processes run on a server equipped with 96 Intel Xeon(R) Gold 5218 CPU cores with 2.30GHz and 160 GB of memory.

6 RESULT AND ANALYSIS

6.1 RQ1: The value of the hyperparameter

We investigate the effectiveness of ReClues for the value of x taking 5, 10, 15, and 20 (i.e., determining the Top-5%, Top-10%, Top-15%, and Top-20% riskiest statements as breakpoints, respectively) on SIR. The results are given in Table 4 and Figure 4.

6.1.1 The capability to estimate the number of faults in different values of x. A promising failure indexing approach should make the number of faults it predicts k equal to the real number of faults r. The values of V_{equal}^{T}, i.e., on how many faulty versions can "$k == r$" be obtained using ReClues equipped with different values of x

Table 4: Comparison in different values of x

T	V^{T}_{equal}	$S\_M^{T}_{FMI}$	$S\_M^{T}_{JC}$	$S\_M^{T}_{PR}$	$S\_M^{T}_{RR}$
$\text{ReClues}_{5\%}$	71	53.47	43.96	54.40	39.67
$\text{ReClues}_{10\%}$	**98**	**73.02**	**59.68**	**74.74**	**55.06**
$\text{ReClues}_{15\%}$	76	55.71	45.13	58.61	44.29
$\text{ReClues}_{20\%}$	71	53.06	43.54	55.16	43.60

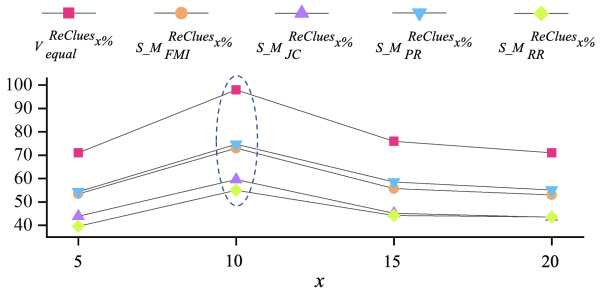


Figure 4: Comparison in different values of x

(T takes $\text{ReClues}_{5\%}$, $\text{ReClues}_{10\%}$, $\text{ReClues}_{15\%}$, and $\text{ReClues}_{20\%}$), are given in Table 4 and Figure 4. It can be seen that when the breakpoint determination threshold is set to 10%, ReClues can correctly estimate the number of faults on 98 faulty versions, while when such a threshold is set to 5%, 15%, and 20%, the numbers of "$k == r$" faulty versions are 71, 76, and 71, respectively.

6.1.2 The capability to cluster in different values of x. For those "$k == r$" faulty versions, we use the four metrics introduced in Section 5.4 to evaluate the clustering effectiveness, as shown in Table 4 and Figure 4. Taking "$S\_M^{ReClues_{10\%}}_{FMI}$: 73.02" as an example. It means that on 98 "$k == r$" faulty versions achieved by $\text{ReClues}_{10\%}$, the sum of the values of FMI, i.e., $FMI_1 + FMI_2 + ... + FMI_{98}$, is 73.02. We can observe that $\text{ReClues}_{10\%}$ delivers 73.02, 59.68, 74.74, and 55.06 points on the four metrics, which is more promising compared with that delivered by the other three variants.

Based on these results, we can find that $\text{ReClues}_{10\%}$ performs best. We think that the Top-5% riskiest statements are not sufficient for representing failures, while the Top-15% and the Top-20% riskiest statements may incur irrelevant information, which can negatively affect failure representation. Therefore, "ReClues" in the next two RQs is "$\text{ReClues}_{10\%}$" in this RQ.

6.2 RQ2: Impact analyses of components

Revisiting Section 4.3, the distance metric of ReClues involves two components, i.e., the breakpoint level and the variable level. In this RQ, we further analyze the impact of each one. To that end, we compare ReClues with its two variants, ReClues_{v^-} (keep only the breakpoint level and ablate the variable level) and ReClues_{b^-} (keep only the variable level and ablate the breakpoint level) on SIR.

As for ReClues_{v^-}, we do not consider original variable information at the breakpoints covered by both two failures, but only consider variables' name. Specifically, if two failures both cover a breakpoint, we simply calculate the value of $Distance^{j}_{var}$ in Formula 2 through dividing the intersection of the names of the variables collected by the two failures at the breakpoint by their union, rather than using Formula 5. And as for ReClues_{b^-}, we merge variable information collected by a failure at all breakpoints into a hunk (without considering coverage of breakpoints), and feed such hunks of two failures into the variable level. That is, we directly use Formula 5 to measure the distance between two failures. The results are given in Figure 5.

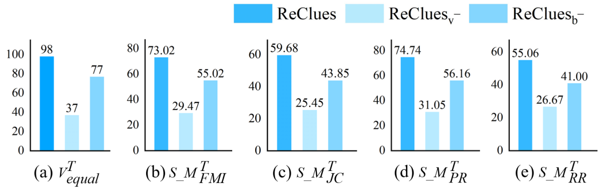


Figure 5: Impact of ReClues components

6.2.1 The capability to estimate the number of faults of different components. Figure 5(a) compares ReClues with its two variants in terms of the capability of faults number estimation, exhibiting performance drops on the condition of an incomplete distance metric. Specifically, ReClues_{v^-} and ReClues_{b^-} can correctly estimate the number of faults on 37 and 77 faulty versions, respectively, decreased by 62.24% and 21.43% respectively compared with ReClues.

6.2.2 The capability to cluster of different components. The remaining four sub-figures in Figure 5 perform the comparison in terms of the capability to cluster, from the perspectives of FMI, JC, PR, and RR. For example. ReClues can get 73.02 points on the metric FMI, while ReClues_{v^-} and ReClues_{b^-} get 29.47 and 55.02 points, respectively. On the metrics JC, PR, and RR, we can observe a similar trend that ReClues_{v^-} and ReClues_{b^-} cause performance degradation.

We can find that the default ReClues performs best, demonstrating that each component in the distance metric positively contributes to ReClues' performance. In particular, ablating the variable level harms the effectiveness of ReClues to a larger extent than ablating the breakpoint level, which double-confirms the intuition of this paper, i.e., the run-time values of program variables can be an effective failure distinguisher in failure indexing.

6.3 RQ3: Competitiveness of ReClues

As we mentioned in Section 2, SD-based and CC-based strategies are the most advanced and prevalent failure proximities to date. Therefore, we compare ReClues with these two for robust and convincing evaluation. Specifically, as for the SD-based proximity, we select MSeer [24] since it is the state-of-the-art in this class. And as for the CC-based proximity, we select Cov_{hit} [28, 47] since it is the most general configuration in this class. Moreover, considering that some works concern the impact of the execution frequency of program statements on debugging [58, 68], we also adopt a variant of Cov_{hit}, i.e., Cov_{count}, which employs execution frequency rather than binary indicators as the fingerprinting function, as a baseline. To evaluate ReClues in a more comprehensive environment, in this RQ, we use both the simulated (SIR) and the real-world (Defects4J) benchmarks. The results are given in Table 5, in which the first four rows (marked as "(S)") depict the results on SIR, while the last four rows (marked as "(D)") depict the results on Defects4J.

Table 5: Comparison with the state-of-the-art techniques

T	V_{equal}^{T}	$S\_M_{FMI}^{T}$	$S\_M_{JC}^{T}$	$S\_M_{PR}^{T}$	$S\_M_{RR}^{T}$
(S) ReClues	**98**	**73.02**	**59.68**	**74.74**	**55.06**
(S) MSeer	68	51.73	42.77	53.25	32.55
(S) Cov$_{count}$	52	41.44	34.57	40.27	27.76
(S) Cov$_{hit}$	29	23.46	20.03	20.46	13.73
(D) ReClues	**37**	**36.92**	**36.85**	**35.92**	**36.12**
(D) MSeer	29	28.99	28.98	28.75	28.75
(D) Cov$_{count}$	24	23.98	23.96	23.50	23.50
(D) Cov$_{hit}$	20	20.00	20.00	20.00	20.00

6.3.1 The capability to estimate the number of faults on SIR. ReClues substantially outperforms all the baseline techniques regarding the capability of faults number estimation. Specifically, ReClues can correctly predict the number of faults on 98 faulty versions on SIR, with 44.12%, 88.46%, and 237.93% improvements compared with MSeer (68), Cov$_{count}$ (52), and Cov$_{hit}$ (29), respectively.

6.3.2 The capability to cluster on SIR. ReClues consistently exceeds three baselines on all the four clustering metrics. For instance, if we focus on the comparison between ReClues and MSeer, improvements are 41.16%, 39.54%, 40.36%, and 69.16%, regarding FMI, JC, PR, and RR, respectively. Considering the four metrics globally, the average improvement is 47.56%. Similarly, in the contexts of comparing ReClues with Cov$_{count}$ and Cov$_{hit}$, the average improvements can be calculated as 83.20% and 243.88%, respectively.

6.3.3 The capability to estimate the number of faults on Defects4J. On all Defects4J faulty versions, ReClues can make k equal to r on 37 faulty versions, it is 27.59%, 54.17%, and 85.00% higher than MSeer (29), Cov$_{count}$ (24), and Cov$_{hit}$ (20), respectively.

6.3.4 The capability to cluster on Defects4J. Similar to that in simulated scenarios, we can also observe that ReClues has a stronger capability of clustering than the baseline techniques in real-world scenarios. In particular, if we focus on the comparison between ReClues and MSeer, improvements are 27.35%, 27.16%, 24.94%, and 25.63%, regarding FMI, JC, PR, and RR, respectively. Considering the four metrics globally, the average improvement is 26.27%. Similarly, when comparing ReClues with Cov$_{count}$ and Cov$_{hit}$, the average improvements are 53.58% and 82.26%, respectively.

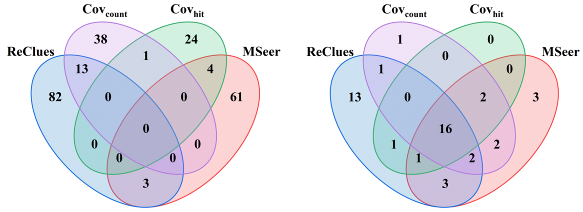
Based on these results, we can find that ReClues significantly outperforms the most advanced and prevalent techniques in the current field of failure indexing, in both simulated and real-world scenarios. Such promising outcomes demonstrate the competitiveness of ReClues, and show the potential of the proposed program variable-based failure proximity.

7 DISCUSSION

7.1 Unique faulty versions handled by ReClues

In the experiments, we find that those "$k == r$" faulty versions achieved by different techniques are not exactly the same. In other words, some faulty versions can only be handled by a certain technique. We further discuss ReClues and the three baselines from this aspect, as shown in Figure 6.

In Figure 6(a), we can find that of the 600 SIR faulty versions, 82 can only be handled by ReClues, 61, 38, and 24 can only be handled by MSeer, Cov$_{count}$, and Cov$_{hit}$, respectively. A similar observation can be drawn from Figure 6(b): of the 100 Defects4J faulty versions, 13 can only be handled by ReClues, 3, 1, and 0 can only be handled by MSeer, Cov$_{count}$, and Cov$_{hit}$, respectively. Though none of the failure indexing techniques is completely dominated by others, there are more faulty versions that can be uniquely handled by ReClues. This result further shows the competitiveness of ReClues from a heuristic perspective, and indicates a potential future direction of combining different failure proximities.



(a) In simulated scenarios (b) In real-world scenarios

Figure 6: The divergence of the "k == r" faulty versions

7.2 Efficiency of ReClues

The time costs of ReClues mainly involve three parts, i.e., the failure representation, the distance measurement, and the clustering. According to our investigation, ReClues typically spends 3.99 minutes and 5.90 minutes on average on generating the proxy for a failed test case, and spends 0.07s and 0.03s on average on measuring the distance between a pair of failed test cases, on SIR and Defects4J faulty versions, respectively. After these two steps are ready, the clustering process typically takes only a few seconds. To summarize, the overhead of ReClues mainly lies in querying variable information at each preset breakpoint.

In fact, failure indexing is essential yet very costly in real-life manual debugging activities, as pointed out by pioneers, *"Experienced developers can manually examine every failure and determine the culprit fault, but this is apparently too expensive"* [47]. In contrast, ReClues can finish this task automatically and hence can save a lot of manual effort, which is much cheaper than manual jobs. It is true that as compared with CC and SD-based methods, ReClues needs higher costs. However, in return, ReClues delivers better performance. Therefore, the cost of ReClues is acceptable: more sophisticated fingerprinting is naturally accompanied by higher overhead, i.e., ***"no free lunch"*** [47].

8 THREATS TO VALIDITY

Our experiments are subject to several threats to validity.

The first is about the representativeness of the benchmark. We evaluate our approach on both simulated and real-world datasets. For the former, we adopt a diversity of mutation operators to inject faults, and for the latter, we collect projects from the industrial programming practice. Although this allows us to have higher confidence with respect to the generalization capability of ReClues, those benchmarks could still not be enough to represent different kinds of software systems. In the future, we plan to further evaluate our approach in larger-scale and more general environments.

The second is about the choice of the evaluation metrics. We select four widely-used metrics, i.e., FMI, JC, PR, and RR, to quantitatively demonstrate the promise of our approach, but all of these four are external metrics, that is, the measurement of the clustering effectiveness is dependent on external information (i.e., the oracle groups). As another type of metrics, internal metrics are based on the generated clusters themselves, which could also contribute to our experimental evaluation, thus mitigating the bias incurred by only employing external metrics. In the future, we plan to integrate them into our work for more robust evaluation.

9 RELATED WORK

As a very early work in this field, Podgurski et al. suggested using code coverage as a failure representer [56]. Since then, such a CC-based failure proximity has been continuously adopted by stakeholders. For example, Huang et al. conducted an empirical study on failure indexing [28] and Wu et al. presented a multi-fault localization technique [74] based on the CC proximity.

Later, Liu and Han regarded two failures as similar if they suggest roughly the same fault location [45]. They introduced a statistical debugging tool [46] to complete the mentioned suggestion process. Such an SD-based failure proximity attracts broad attention from academia. For example, Jones et al. utilized Tarantula [32, 33], while Gao and Wong utilized Crosstab [24, 71], to facilitate the fault location suggestion in the SD proximity.

If the coverage of the failures having different root causes is identical, neither the CC nor the SD-based tactic can work well. This paper utilizes the run-time values of program variables for getting rid of this bottleneck, showing a remarkable improvement.

There are some recent works that introduce external profiles to support the failure indexing, such as code-independent features in regression testing [26], as well as code features and historical features in continuous integration [2]. We do not consider such types of studies since they go beyond our research scope: 1) their source information cannot be always available, and 2) this paper focuses on failure indexing in the context of multi-fault localization.

10 CONCLUSION

In this paper, we propose a novel type of failure proximity, namely, the program variable-based failure proximity, and further present ReClues, a variable information-based failure indexing approach. ReClues mainly comprises the newly-defined fingerprinting function that integrates the run-time values of program variables to represent failures, and the distance metric designed to cooperate with the fingerprinting function. Experiments demonstrate the competitiveness of ReClues. Specifically, compared with the state-of-the-art technique, ReClues can achieve 44.12% and 27.59% improvements in faults number estimation, as well as 47.56% and 26.27% improvements in clustering effectiveness, in simulated and real-world environments, respectively. Besides, there are more faulty programs that can only be handled by ReClues compared with using the other techniques in our experiment.

In the future, we plan to draw on deep learning methods to deliver a stronger failure indexing approach. A further trial with larger and more general benchmarks as well as a broader spectrum of evaluation metrics is also being conceived.

ACKNOWLEDGMENT

This work was partially supported by the National Natural Science Foundation of China under the grant numbers 62250610224, 61972289, and 61832009. And the numerical calculations in this work have been partially done on the supercomputing system in the Supercomputing Center of Wuhan University.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 88–99.

[2] Gabin An, Juyeon Yoon, Jeongju Sohn, Jingun Hong, Dongwon Hwang, and Shin Yoo. 2022. Automatically Identifying Shared Root Causes of Test Breakages in SAP HANA. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice*. 65–74.

[3] Gabin An, Juyeon Yoon, and Shin Yoo. 2021. Searching for multi-fault programs in defects4j. In *International Symposium on Search Based Software Engineering*. Springer, 153–158.

[4] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th International Conference on Software Engineering*. 402–411.

[5] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[6] Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2018. Spectrum-based fault localization in software product lines. *Information and Software Technology* 100 (2018), 18–31.

[7] Arun Babu, Qingkai Shi, and Muhammad Ashfaq. 2020. Python script for performing mutation testing. Github Repository. https://github.com/arun-babu/mutate.py

[8] Antonia Bertolino, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2017. Adaptive coverage and operational profile-based testing for reliability improvement. In *2017 IEEE/ACM 39th International Conference on Software Engineering*. IEEE, 541–551.

[9] He-ling CAO and Shu-juan JIANG. 2017. Multiple-fault localization based on chameleon clustering. *ACTA ELECTONICA SINICA* 45, 2 (2017), 394.

[10] Stephen L Chiu. 1994. Fuzzy model identification based on cluster estimation. *Journal of Intelligent & Fuzzy Systems* 2, 3 (1994), 267–278.

[11] Vidroha Debroy and W Eric Wong. 2009. Insights on fault interference for programs with multiple bugs. In *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 165–174.

[12] Nicholas DiGiuseppe and James A Jones. 2011. Fault interaction and its repercussions. In *2011 27th IEEE International Conference on Software Maintenance*. IEEE, 3–12.

[13] Nicholas DiGiuseppe and James A Jones. 2011. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 210–220.

[14] Nicholas DiGiuseppe and James A Jones. 2012. Concept-based failure clustering. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–4.

[15] Nicholas DiGiuseppe and James A Jones. 2012. Software behavior and failure clustering: An empirical study of fault causality. In *2012 IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 191–200.

[16] Nicholas DiGiuseppe and James A Jones. 2015. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering* 20, 4 (2015), 928–967.

[17] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

[18] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 32, 9 (2006), 733–752.

[19] Nicolas Dupin and Frank Nielsen. 2023. Partial K-Means with M Outliers: Mathematical Programs and Complexity Results. In *International Conference on Optimization and Learning*. Springer, 287–303.

[20] Yang Feng, James Jones, Zhenyu Chen, and Chunrong Fang. 2018. An empirical study on software failure classification with multi-label and problem-transformation techniques. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*. IEEE, 320–330.

[21] Abby Flynt and Yipeng Huang. 2018. Exploration of common clustering methods and the behavior of certain performance indices. *Ball State Undergraduate Mathematics Exchange* 12, 1 (2018), 35.

[22] Wei Fu and Patrick O Perry. 2020. Estimating the number of clusters using cross-validation. *Journal of Computational and Graphical Statistics* 29, 1 (2020),

162–173.

[23] Meng Gao, Pengyu Li, Congcong Chen, and Yunsong Jiang. 2018. Research on software multiple fault localization method based on machine learning. In *MATEC Web of Conferences*, Vol. 232. EDP Sciences, 01060.

[24] Ruizhi Gao and W Eric Wong. 2019. MSeer—An Advanced Technique for Locating Multiple Bugs in Parallel. *IEEE Transactions on Software Engineering* 45, 03 (2019), 301–318.

[25] Laleh Sh Ghandehari, Yu Lei, Raghu Kacker, Richard Kuhn, Tao Xie, and David Kung. 2018. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering* 46, 6 (2018), 616–645.

[26] Mojdeh Golagha, Constantin Lehnhoff, Alexander Pretschner, and Hermann Ilmberger. 2019. Failure clustering without coverage. In *Proceedings of the 28th International Symposium on Software Testing and Analysis*. 134–145.

[27] Wolfgang Högerle, Friedrich Steimann, and Marcus Frenkel. 2014. More debugging in parallel. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 133–143.

[28] Yanqin Huang, Junhua Wu, Yang Feng, Zhenyu Chen, and Zhihong Zhao. 2013. An empirical study on clustering for isolating bugs in fault localization. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 138–143.

[29] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[30] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. 1999. Data clustering: a review. *Comput. Surveys* 31, 3 (1999), 264–323.

[31] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. 167–178.

[32] James A Jones, James F Bowring, and Mary Jean Harrold. 2007. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. 16–26.

[33] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 273–282.

[34] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*. IEEE, 467–477.

[35] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[36] 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*. 654–665.

[37] Leonard Kaufman and Peter J. Rousseeuw. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley InterScience.

[38] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 114–125.

[39] Maurice George Kendall. 1948. Rank correlation methods. (1948).

[40] Suneel Kumar Kingrani, Mark Levene, and Dell Zhang. 2018. Estimating the number of clusters using diversity. *Artificial Intelligence Research* 7, 1 (2018), 15–22.

[41] Si-Mohamed Lamraoui and Shin Nakajima. 2016. A formula-based approach for automatic fault localization of multi-fault programs. *Journal of Information Processing* 24, 1 (2016), 88–98.

[42] Michael Levandowsky and David Winter. 1971. Distance between sets. *Nature* 234, 5323 (1971), 34–35.

[43] Zheng Li, Yonghao Wu, and Yong Liu. 2019. An empirical study of bug isolation on the effectiveness of multiple fault localization. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security*. IEEE, 18–25.

[44] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. 2006. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering* 32, 10 (2006), 831–848.

[45] Chao Liu and Jiawei Han. 2006. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 46–56.

[46] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. 2005. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 286–295.

[47] Chao Liu, Xiangyu Zhang, and Jiawei Han. 2008. A systematic study of failure proximity. *IEEE Transactions on Software Engineering* 34, 6 (2008), 826–843.

[48] Chao Liu, Xiangyu Zhang, Jiawei Han, Yu Zhang, and Bharat K Bhargava. 2007. Indexing noncrashing failures: A dynamic program slicing-based approach. In *2007 IEEE International Conference on Software Maintenance*. IEEE, 455–464.

[49] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89 (2014), 51–62.

[50] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[51] Sumathi Pawar, H Manjula Gururaj, and Niranajan N Chiplunar. 2022. Text Summarization Using Document and Sentence Clustering. *Procedia Computer Science* 215 (2022), 361–369.

[52] Spencer Pearson. 2016. Evaluation of fault localization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 1115–1117.

[53] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering*. IEEE, 609–620.

[54] Hanyu Pei, Beibei Yin, Min Xie, and Kai-Yuan Cai. 2021. Dynamic random testing with test case clustering and distance-based parameter adjustment. *Information and Software Technology* 131 (2021), 106470.

[55] Francesco Piccialli, Salvatore Cuomo, Vincenzo Schiano di Cola, and Giampaolo Casolla. 2019. A machine learning approach for IoT cultural data. *Journal of Ambient Intelligence and Humanized Computing* (2019), 1–12.

[56] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 465–475.

[57] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[58] Ting Shu, Tiantian Ye, Zuohua Ding, and Jinsong Xia. 2016. Fault localization based on statement frequency. *Information Sciences* 360 (2016), 43–56.

[59] Yi Song, Xiaoyuan Xie, Xihao Zhang, Quanming Liu, and Ruizhi Gao. 2022. Evolving Ranking-Based Failure Proximities for Better Clustering in Fault Isolation. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[60] Friedrich Steimann and Marcus Frenkel. 2012. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 121–130.

[61] Shih-Feng Sun and Andy Podgurski. 2016. Properties of effective metrics for coverage-based statistical fault localization. In *2016 IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 124–134.

[62] Robert Tibshirani, Guenther Walther, and Trevor Hastie. 2001. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63, 2 (2001), 411–423.

[63] Jingxuan Tu, Xiaoyuan Xie, and Baowen Xu. 2016. Code coverage-based failure proximity without test oracles. In *2016 IEEE 40th Annual Computer Software and Applications Conference*, Vol. 1. IEEE, 133–142.

[64] Jeffrey M. Voas. 1992. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering* 18, 8 (1992), 717.

[65] Qing Wang, Shujian Wu, and Ming-Shu Li. 2008. Software defect prediction. *Journal of Software* 19, 7 (2008), 1565–1580.

[66] Xingya Wang, Shujuan Jiang, Pengfei Gao, Kai Lu, Bo Lili, Xiaolin Ju, and Yanmei Zhang. 2020. Fuzzy C-Means Clustering Based Multi-Fault Localization. *Chinese Journal of Computers* 43, 2 (2020), 206–232.

[67] Yabin Wang, Ruizhi Gao, Zhenyu Chen, W Eric Wong, and Bin Luo. 2014. WAS: A weighted attribute-based strategy for cluster test selection. *Journal of Systems and Software* 98 (2014), 44–58.

[68] Wanzhi Wen. 2012. Software fault localization based on program slicing spectrum. In *2012 34th International Conference on Software Engineering*. IEEE, 1511–1514.

[69] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus Agus Haryono, Shaowei Wang, and David Lo. 2022. Real world projects, real faults: evaluating spectrum based fault localization techniques on Python projects. *Empirical Software Engineering* 27, 6 (2022), 147.

[70] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[71] W Eric Wong, Vidroha Debroy, and Dianxiang Xu. 2011. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 3 (2011), 378–396.

[72] 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.

[73] Junjie Wu, Hui Xiong, and Jian Chen. 2009. Adapting the right measures for k-means clustering. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 877–886.

[74] Yong-Hao Wu, Zheng Li, Yong Liu, and Xiang Chen. 2020. Fatoc: Bug isolation based multi-fault localization by using optics clustering. *Journal of Computer Science and Technology* 35, 5 (2020), 979–998.

[75] Juanying Xie, Ying Zhou, Mingzhao Wang, and Weiliang Jiang. 2017. New criteria for evaluating the validity of clustering. *CAAI Transactions on Intelligent Systems*

12, 6 (2017), 873–882.

[76] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology* 22, 4 (2013), 1–40.

[77] Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. 2010. Isolating suspiciousness from spectrum-based fault localization techniques. In *2010 10th International Conference on Quality Software*. IEEE, 385–392.

[78] Xiaoyuan Xie, W Eric Wong, Tsong Yueh Chen, and Baowen Xu. 2013. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology* 55, 5 (2013), 866–879.

[79] Xiaofeng Xu, Vidroha Debroy, W Eric Wong, and Donghui Guo. 2011. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering* 21, 06 (2011), 803–827.

[80] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[81] Xiaozhen Xue and Akbar Siami Namin. 2013. How significant is the effect of fault interactions on coverage-based fault localizations?. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 113–122.

[82] Ronald R Yager and Dimitar P Filev. 1994. Approximate clustering via the mountain method. *IEEE Transactions on systems, man, and Cybernetics* 24, 8 (1994), 1279–1284.

[83] Xiaobo Yan, Bin Liu, and Shihai Wang. 2018. An analysis on the negative effect of multiple-faults for spectrum-based fault localization. *IEEE Access* 7 (2018), 2327–2347.

[84] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2017. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology* 26, 1 (2017), 1–30.

[85] Zhongxing Yu, Chenggang Bai, and Kai-Yuan Cai. 2015. Does the failing test execute a single or multiple faults? An approach to classifying failing tests. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 924–935.

[86] Abubakar Zakari, Sai Peck Lee, and Ibrahim Abaker Targio Hashem. 2019. A community-based fault isolation approach for effective simultaneous localization of faults. *IEEE Access* 7 (2019), 50012–50030.

[87] Lejun Zhang, Jinlong Wang, Weizheng Wang, Zilong Jin, Yansen Su, and Huiling Chen. 2022. Smart contract vulnerability detection combined with multi-objective detection. *Computer Networks* 217 (2022), 109289.