The Journal of Systems & Software 193 (2022) 111452

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A comprehensive empirical investigation on failure clustering in parallel debugging ☆

Yi Song, Xiaoyuan Xie *, Quanming Liu, Xihao Zhang, Xi Wu

School of Computer Science, Wuhan University, China

ARTICLE INFO

Article history:
Received 24 November 2021
Received in revised form 13 July 2022
Accepted 15 July 2022
Available online 26 July 2022

Keywords:
Failure clustering
Fault isolation
Multiple-fault
Parallel debugging

ABSTRACT

The clustering technique has attracted a lot of attention as a promising strategy for parallel debugging in multi-fault scenarios, this heuristic approach (i.e., failure indexing or fault isolation) enables developers to perform multiple debugging tasks simultaneously through dividing failed test cases into several disjoint groups. When using statement ranking representation to model failures for better clustering, several factors influence clustering effectiveness, including the risk evaluation formula (REF), the number of faults (NOF), the fault type (FT), and the number of successful test cases paired with one individual failed test case (NSP1F). In this paper, we present the first comprehensive empirical study of how these four factors influence clustering effectiveness. We conduct extensive controlled experiments on 1060 faulty versions of 228 simulated faults and 141 real faults, and the results reveal that: (1) GP19 is highly competitive across all REFs, (2) clustering effectiveness decreases as NOF increases, (3) higher clustering effectiveness is easier to achieve when a program contains only predicate faults, and (4) clustering effectiveness remains when the scale of NSP1F is reduced to 20%.

1. Introduction

Programs often produce unexpected results that deviate from oracles during software testing, such anomalous behavior indicates that at least one fault resides in the program. However, locating these faults is generally labor-intensive and tedious in the debugging process (Wong et al., 2016; Xiaobo et al., 2018). Generally, in multi-fault scenarios, there are two commonly adopted strategies:

- **Sequential debugging.** Ignoring the linkage between failed test cases and faults, this strategy detects, localizes, and fixes one fault, and then reruns the test suite (TS, which contains all test cases) on the semi-repaired program under test (PUT) again, iterates these steps until a failure-free program is delivered.
- **Parallel debugging.** This strategy first mines the linkage that exists between failed test cases and faults, that is, divides all failed test cases into several disjoint fault-focused clusters through clustering techniques (with the goal of the failed test cases in a cluster to be triggered by the same root cause, and the failed test cases in different clusters to be triggered by different root causes), and combines each fault-focused cluster with all successful test cases to form several fault-focused TS, finally assigns them to different developers for parallel localization (Jones et al., 2007).

Many empirical studies have shown that sequential debugging does not perform well in localizing multiple faults (DiGiuseppe and Jones, 2011a,b, 2015), while parallel debugging shows promise in this area. The core of parallel debugging lies in clustering. Only by properly capturing the linkage between failed test cases and faults, as well as heuristically dividing failed test cases, can a hunk of localization task be decomposed into several sub-tasks with high quality. However, most previous research in terms of parallel debugging concentrated on the localization process after clustering, with only a few studies investigating the clustering process, one of the most critical steps that may affect the overall parallel debugging performance. Several factors may affect the failure clustering step, but there is a lack of comprehensive empirical studies investigating these variables.

Therefore, in this paper, we conduct the first comprehensive empirical investigation, aiming at the clustering step by selecting four factors that could influence clustering effectiveness: the risk evaluation formula (REF) that represents failed test cases, the number of faults (NOF) and the fault type (FT) contained in the program, and the number of successful test cases paired with one individual failed test case (NSP1F), and further proposing four research questions as follows to guide our extensive experiments (these abbreviations are listed in Table 1 for easy tracking).

☆ Editor: Matthias Galster.

* Corresponding author.
E-mail address: xxie@whu.edu.cn (X. Xie).

https://doi.org/10.1016/j.jss.2022.111452
0164-1212/

Table 1
Abbreviations and their full forms.

Abbreviations	Full forms
REF	Risk Evaluation Formula
NOF	The Number Of Faults
FT	The Fault Type
NSP1F	The Number of Successful test cases Paired with ONE Failed test case
TS	Test Suite
PUT	Program Under Test
AF	Assignment Fault
PF	Predicate Fault

• **RQ1: Do different REFs have the same capability to representing failed test cases?**

Failed test cases are typically too unstructured and abstract to be used directly for clustering. Many approaches, such as coverage vector representation (CVR) and statement ranking representation (SRR), have been utilized to convert failed test cases into structured and mathematical forms. CVR is similar to T-proximity (Trace-proximity) in Liu et al. (2008), in which a vector with a length equal to the number of executable statements in PUT is created to represent a failed test case, with the value of the *i*th element being set to 1 if this failed test case covers the *i*th statement, and 0 otherwise. SRR is similar to R-proximity (Rank-proximity) in Liu et al. (2008), in which one failed test case and successful test cases are executed on PUT, and the coverage information of the program execution is collected and organized in the form of notations defined in spectrum-based fault localization (SBFL) (Xie and Xu, 2021). The coverage is then input into an REF to produce a ranking list that reflects statements' suspiciousness, which is employed to represent this failed test case finally. SRR has been proved to be superior to CVR in representing failed test cases (Liu et al., 2008), which has also been adopted by a number of previous research due to its advantage in translating a failed test case into a clustering-friendly proxy (Cao and Jiang, 2017; Yu et al., 2015; Wang et al., 2014; Gao and Wong, 2019).

In SRR, REF is used to produce a ranking list that contains the execution features of a failed test case. Obviously, a better REF should extract more discriminative features for failed test cases caused by different root causes, in other words, the distance between ranking lists that represent failed test cases triggered by different faults should be greater than the distance between ranking lists that represent failed test cases triggered by a same fault. However, almost all existing studies only simply chose a specific REF to generate the ranking list. To the best of our knowledge, no research has contrasted the capabilities of various REFs in representing failed test cases. To that end, we analyze 35 commonly-used REFs through extensive experiments in this RQ from this perspective.

• **RQ2: How NOF affects clustering effectiveness?**

Although it is difficult to know whether a faulty program contains a single fault or multiple faults exactly, we can intuitively infer the more faults it has, the more effort and time the debugging process will take (DiGiuseppe and Jones, 2011b; Xue and Namin, 2013). Many studies have investigated the effect of NOF on the effectiveness of fault localization techniques (DiGiuseppe and Jones, 2011b, 2015; Jones et al., 2002), but few have explored the influence of NOF on the clustering process. We analyze how clustering effectiveness changes as NOF grows in 2-bug, 3-bug, 4-bug, and 5-bug scenarios (i.e., programs that contain 2, 3, 4, and 5 bugs, respectively).

• **RQ3: Is clustering effectiveness affected by FT?**

In addition to NOF, FT is also an essential factor in the debugging process. Although the randomness and uncertainty of the programming process determine the diversity of the introduced faults, the most common FTs typically refer to assignment faults (Jeffrey et al., 2008) and predicate faults (Xuan et al., 2016). If a program has only assignment faults, only predicate faults, or both of them, how will the clustering effectiveness be affected? We discuss each of the three scenarios separately.

• **RQ4: Will clustering effectiveness be reduced using a lower NSP1F?**

When using SRR to represent failed test cases, almost all researchers pair one individual failed test case with all successful test cases (Cao and Jiang, 2017; Yu et al., 2015) without giving any reason or explaining the rationality behind this strategy. If one failed test case is paired with *part of* rather than *all* successful test cases, the cost of debugging will be probably reduced, but will this reduction harm clustering effectiveness? We contrast the clustering effectiveness in five scenarios by pairing a failed test case with X percent of successful test cases ($X = 100$, 80, 60, 40, 20).

Furthermore, the distance metric, the estimation of the number of clusters and the assignment of initial medoids, as well as the clustering algorithm are also critical factors in determining clustering effectiveness in parallel debugging. Gao and Wong have proposed a parallel debugging approach, MSeer (Gao and Wong, 2019), to solve the aforementioned concerns. In particular, they revised the traditional Kendall tau distance (Kendall and Gibbons, 1990), presented an innovative strategy to assign initial medoids during predicting the number of clusters based on the mountain method (Yager and Filev, 1994; Chiu, 1994), and refined the K-medoids clustering algorithm (Kaufman and Rousseeuw, 2009). We will discuss our four research questions and conduct experiments based on MSeer due to its innovation and high effectiveness. A further introduction regarding MSeer is given in Section 2.3.

We create 1060 faulty versions of nine programs, *flex*, *grep*, *gzip*, *sed*, *Chart*, *Closure*, *Lang*, *Math*, and *Time*, as our benchmark. The experimental results show that[1]:

(1) GP19 (the 19th formula evolved by Genetic Programming in Yoo (2012)) is highly competitive across all REFs when representing failed test cases.
(2) Clustering effectiveness decreases as NOF grows.
(3) Higher clustering effectiveness is easier to achieve when a faulty program contains only predicate faults.
(4) Clustering effectiveness remains when NSP1F is reduced to 20%.

The main contributions of this paper are as follows:

(1) Unlike previous studies that contrasted REFs from the perspective of fault localization effectiveness, we contrast 35 REFs (including the latest Crosstab, Dstar, and GP02, GP03, GP19 evolved by genetic programming) in terms of how well they represent failed test cases. We recommend GP19, an REF with strong competitiveness in extracting failed test cases' execution features for future researchers.
(2) Our controlled experiments reveal that the effectiveness of clustering failed test cases will reduce when NOF increases.
(3) We analyze two typical types of faults, assignment faults and predicate faults, and discover that it is easier to achieve higher clustering effectiveness when a program contains only predicate faults.
(4) We pair 100%, 80%, 60%, 40%, and 20% of successful test cases with one failed test case, and contrast the clustering effectiveness in these five scenarios. The findings indicate

[1] The replication package of this empirical study is available at this website.

that cutting the scale of successful test cases has little effect on clustering effectiveness, suggesting a way worth trying to lower the cost of SRR representation for future researchers.

The remainder of this paper is organized as follows: Section 2 introduces the background knowledge. Section 3 describes the experimental dataset and setup. Section 4 analyzes the experimental results. Section 5 discusses some interesting topics. Section 6 is the threats to validity. Section 7 reports related works. Conclusions and directions for future work are proposed in Section 8.

2. Background

We explain why clustering failed test cases is essential and present the rationale of parallel debugging in Section 2.1. The principles and technical details of SRR are given in Section 2.2, followed by a motivating example showing the application of SRR-based failure clustering in Section 2.3.

2.1. Why clustering?

In general, the possibility of a program being faulty and the number of faults it contains are proportional to its size (Wang et al., 2008). With the increasing volume and the explosive growth of code in modern software systems, most faulty programs usually have multiple faults.

In multi-fault scenarios, various failed test cases[2] may be caused by different faults. If failed test cases with distinct root causes are not divided properly, fault localization techniques could be confused by the impure test suite significantly, for example, SBFL techniques extract execution features of all faults guided by the impure spectrum information, which will lower the rank of each fault in the generated ranking list. According to Wang et al. *failed test cases that are not related to specific fault are the main reason to reduce the effectiveness of SBFL* (Wang et al., 2020), and similarly, Keller et al. have drawn a similar conclusion, *when using SBFL techniques, the number of lines that need to be inspected can be reduced by high quality test cases that execute the bug* (Keller et al., 2017). Therefore, the purpose of dividing failed test cases in a multi-fault scenario is to allow failed test cases with different root causes to target their corresponding faults separately, to put it another way, reduce the interferences among multiple faults in a program, enhance the pertinence of fault localization techniques and thus achieve parallel debugging.

Many researchers have attempted to employ the clustering technique to divide failed test cases (Jones et al., 2007; Gao and Wong, 2019; Wu et al., 2020; Golagha et al., 2019; DiGiuseppe and Jones, 2012). Ideally, failures caused by the same fault should be grouped into a cluster, then the failed test cases in a cluster are combined with all successful test cases to form a fault-focused TS targeting a specific fault, as defined in Formula (1) and Formula (2). This strategy is often called failure indexing or fault isolation.

$$F_t = F_1 \cup F_2 \cup \cdots \cup F_r \tag{1}$$

$$\textit{fault-focused } TS_i = F_i \cup S (i = 1, 2, \ldots, r) \tag{2}$$

where F_t and S represent all failed test cases and all successful test cases in TS, respectively. F_1, F_2, …, F_r are generated fault-focused clusters, and r is the number of clusters (which is expected to be equal to the number of faults).

[2] In this paper, we use "failed test case", "anomalous execution", and "failure" interchangeably.

Table 2
Notations in spectrum information.

Notation	Meaning
N_{CF}	The number of failed test cases covering a statement
N_{UF}	The number of failed test cases not covering a statement
N_{CS}	The number of successful test cases covering a statement
N_{US}	The number of successful test cases not covering a statement
N_C	The number of test cases covering a statement
N_U	The number of test cases not covering a statement
N_S	Total number of successful test cases
N_F	Total number of failed test cases
N	Total number of test cases

Clustering failed test cases is a heuristic strategy for improving the pertinence of TS and the effectiveness of fault localization, this widely acknowledged method has been adopted by many previous studies in the field of multi-fault localization (Gao and Wong, 2019; Podgurski et al., 2003; Steimann and Frenkel, 2012).

It is vital to encode failed test cases in an intermediate representation due to their unfriendly form for clustering. Currently, the most widely used representation methods are aforementioned CVR and SRR. The technical details of SRR, which is employed to conduct experiments in this paper, are described below.

2.2. Statement ranking representation

After TS have been executed on PUT, the coverage information of each test case that contains two components will be collected in SRR:

- **Execution Path**: A binary vector that records which program entities (statements,[3] branches, functions, or basic blocks) (Reps et al., 1997; Harrold et al., 2000) have been covered by a test case.
- **Execution result**: A binary value denotes whether or not the actual output of a test case matches its expected output.

Suppose there is a PUT containing j executable statements s_i ($i = 1, 2, \ldots, j$) and a TS containing p test cases t_i ($i = 1, 2, \ldots, p$), the coverage generated by running TS against PUT should be a matrix of size $j \times p$. In SRR, the coverage gathered against a failed test case and successful test cases will be converted into spectrum information according to the notations defined in SBFL (Xie and Xu, 2021), as shown in Table 2.[4]

To incorporate several notations in spectrum information into a suspiciousness value that measures the risk of a statement being faulty, researchers have constructed a series of risk evaluation formulas. For example, Ochiai proposed by Abreu et al. is defined in Formula (3) (Abreu et al., 2006):

$$\textit{suspiciousness}_{\text{Ochiai}} = \frac{N_{CF}}{\sqrt{N_F N_C}} \tag{3}$$

The statements[5] in PUT are ranked according to their suspiciousness in descending order to deliver a ranking list. This type of ranking list, which is produced by an REF from spectrum information that reflects the execution features of a failed test case and successful test cases, is employed to represent this failed test case in SRR.

2.3. Motivating example

The workflow of SRR-based failure clustering is illustrated in Fig. 1. Test cases in the test suite can be determined as failed or

[3] We implement the statement granularity in our experiments, hence "entity" and "statement" are interchangeable hereafter.

[4] Also referred to as a_{ef}, a_{nf}, a_{ep}, a_{np}, a_e, a_n, a_p, a_f, a, respectively.

[5] Unless otherwise specified, "statement" refers to "executable statement" in this paper.

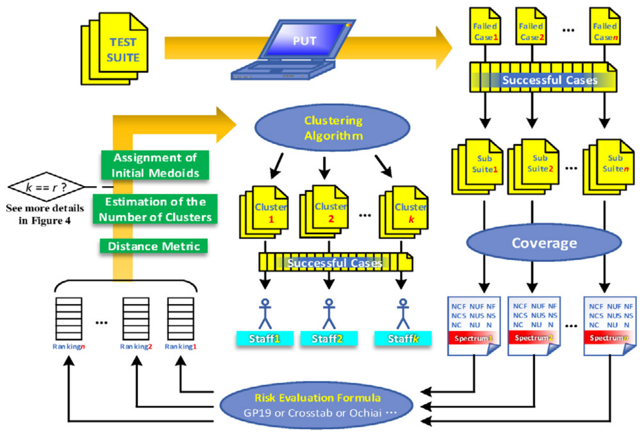


Fig. 1. The workflow of SRR-based failure clustering.

successful after being executed against the program, according to the inconsistency or consistency between actual and expected outputs, respectively. Each of failed test cases will be combined with successful test cases and then be fed into a risk evaluation formula, for delivering a ranking list that could represent it in a mathematical form. Once fault-focused clusters are produced by clustering these ranking lists, they will be immediately sent to different handlers for the following step. It should be noted that after failed test cases have been transformed to ranking lists, it is necessary to preprocess such data by measuring distances between them, estimating the number of clusters, and assigning the initial medoids, and only after all of these procedures have been fulfilled can the clustering algorithm begin to work. MSeer, an advanced framework for localizing multiple faults in parallel that alleviated these challenging jobs, has been proposed by Gao and Wong (2019). Specifically, they (1) claimed that in the classic Kendall tau distance metric, discordant pairs of more suspicious statements should contribute more to the distance between two ranking lists, and proposed a modified distance metric based on this intuition; (2) assigned a potential value to each of failed test cases (ranking lists) based on data winsorization, and developed an algorithm to judge whether a failed test case should be chosen as one of medoids; (3) relieved the shortcoming of examining all possible combinations of data points as initial medoids that exists in the traditional K-medoids clustering algorithm. We conduct our experiments based on MSeer because it has been recognized as one of the state-of-the-art parallel debugging techniques, along with its availability and reliability.

Let us use a motivating example to illustrate the details of SRR as well as demonstrate the promise of failure clustering. As shown in Table 3, the PUT that contains 11 statements, is designed to calculate the product of the smaller two of the three numbers, in which two faults have been induced by statements s_6 and s_9, respectively. Give a TS containing 10 test cases: t_1 = {1,2,4}, t_2 = {4,3,2}, t_3 = {3,2,4}, t_4 = {5,1,6}, t_5 = {2,6,5}, t_6 = {6,5,1}, t_7 = {7,5,8}, t_8 = {5,7,3}, t_9 = {8,1,2}, t_{10} = {8,6,9}, six of them are labeled as *failed* due to the unexpected outputs (t_3, t_4, t_5, t_7, t_8, t_{10}). The 11 × 10 matrix composed of rows s_1 to s_{11} and columns t_1 to t_{10} in Table 3 is the coverage obtained by running TS against PUT, where $t_1 \sim t_{10}$ columns represent the execution paths of 10 test cases. The symbol "·" denotes that a test case covers an innocent statement, while "▲" and "△" denote that a test case covers the statements containing $Fault_1$ and $Fault_2$, respectively. The coverage information is reorganized to spectrum information according to the notations defined in Table 2, as shown in the 11 × 9 matrix composed of rows s_1 to s_{11} and columns N_{CF} to N in Table 3.

Each statement's suspiciousness is then generated by Ochiai, as shown in column $F_t \cup S$ in Table 3. We can immediately sort these statements in descending order of suspiciousness, and then get a ranking list of them: s_9, s_1, s_2, s_7, s_8, s_5, s_6, s_3, s_4, s_{10}, s_{11}. The statement s_9 containing $Fault_2$ has the highest suspiciousness of 0.82, hence it will be inspected first. However, the statement s_6 containing $Fault_1$ is ranked seventh, innocent statements s_1, s_2, s_7, s_8 and s_5 will be examined before s_6. This simple example reveals that the impure TS has a limited capability to delivering a promising fault localization output.

Now we depict how fault localization effectiveness will be improved by grouping failed test cases into distinct fault-focused clusters. This is also a step-by-step elaboration of Fig. 1.

- **For the failure representation.** We employ SRR to represent all six failed test cases. Take t_5 as an example. Pairing t_5 with S to form a failure-specific TS, $t_5 \cup S$, executing this TS on PUT to obtain coverage and convert it into spectrum information,[6] and then utilizing a risk evaluation formula (e.g., Ochiai) to incorporate the spectrum information for obtaining each statement's suspiciousness, finally, a ranking list can be produced to represent t_5, as shown in Table 4, which will be invoked in the subsequent clustering process as a proxy of t_5. It should be noted that there are many ways for producing a ranking list according to statements' suspiciousness (Huang et al., 2013). Considering the intuition that a ranking list should clearly reflect the priority of a statement being inspected, as well as other previous studies' experience (Huang et al., 2013), we adopt the following ranking strategy: if several statements with the same suspiciousness form a *Tie* (Xu et al., 2011), the rankings of all statements in the *Tie* will be set to the beginning position of this *Tie*.
- **For the distance metric.** Given two ranking lists that represent failed test cases, the classical Kendall tau distance counts the number of pairwise disagreements between them. Considering the characteristic of ranking lists in the context of failure representation, discordant pairs of more risky statements (i.e., at lower positions in the ranking lists) should be paid more attention. Based on this intuition, we use the revised Kendall tau distance, which takes the reciprocal of the position of statements in the discordant pairs (Gao and Wong, 2019), to measure the similarity between each pair of failed test cases.
- **For the estimation of the number of clusters and the assignment of initial medoids.** We assign a potential value for each failed test case according to the density of its surrounding, to reflect the possibility of it being set as a medoid, and the failed test case with the highest potential value will be selected as the first medoid. Then, all failed test cases' potential values will be updated based on how far they are from the newest medoid. Repeating these steps iteratively until the highest potential value falls within a predefined threshold, and as a consequence of which, the number of clusters and initial medoids can be determined at the same time (Gao and Wong, 2019).
- **For the clustering algorithm.** The K-medoids clustering approach sets practical (not virtual) data points as medoids, aiming at minimizing the distance between failed test cases and the medoid of the cluster where they reside. Its traditional version suffers from two tricky problems, namely, the difficulty of choosing a proper distance metric and the overhead caused by examining all possible combinations of data

[6] This failure-specific TS's coverage and the corresponding spectrum information are omitted due to limited space.

Table 3
The sample PUT and its coverage against the given TS.

S	Program	Coverage information										Spectrum information									Suspiciousness		
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	N_{CF}	N_{UF}	N_{CS}	N_{US}	N_C	N_U	N_S	N_F	N	$F_t \cup S$	$F_1 \cup S$	$F_2 \cup S$
s_1	input a, b, c	·	·	·	·	·	·	·	·	·	·	6	0	4	0	10	0	4	6	10	0.77	0.58	0.71
s_2	if (a < b):	·	·	·	·	·	·	·	·	·	·	6	0	4	0	10	0	4	6	10	0.77	0.58	0.71
s_3	if (b < c):	·				·			·			2	4	1	3	3	7	4	6	10	0.47	0.82	0
s_4	z = a * b	·										0	6	1	3	1	9	4	6	10	0	0	0
s_5	else:					·			·			2	4	0	4	2	8	4	6	10	0.58	**1**	0
s_6	z = b * c //*Fault*$_1$ ✓z = a * c					▲			▲			2	4	0	4	2	8	4	6	10	0.58	**1**	0
s_7	else:		·	·	·		·	·		·	·	4	2	3	1	7	3	4	6	10	0.62	0	0.76
s_8	if (a < c)		·	·	·		·	·		·	·	4	2	3	1	7	3	4	6	10	0.62	0	0.76
s_9	z = a * c //*Fault*$_2$ ✓z = a * b			△	△			△			△	4	2	0	4	4	6	4	6	10	**0.82**	0	**1**
s_{10}	else		·				·			·		0	6	3	1	3	7	4	6	10	0	0	0
s_{11}	z = b * c		·				·			·		0	6	3	1	3	7	4	6	10	0	0	0

Table 4
Statements' suspiciousness calculated by Ochiai in the sample PUT against $t_5 \cup S$ and the corresponding ranking list.

Statement	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}
Suspiciousness	0.45	0.45	0.71	0	1	1	0	0	0	0	0
Ranking list	4	4	3	6	1	1	6	6	6	6	6

samples as initial medoids. The aforementioned two strategies can properly handle these two points, respectively, thus an improved K-medoids algorithm can be delivered and used in our failure clustering (Gao and Wong, 2019). In the motivating example, failed test cases t_5 and t_8 are triggered by $Fault_1$, and t_3, t_4, t_7, and t_{10} are triggered by $Fault_2$. Ideally, the clustering results should be $F_1 = \{t_5, t_8\}$, $F_2 = \{t_3, t_4, t_7, t_{10}\}$.[7]

- **For the bug triage.** Two fault-focused TSs, $F_1 \cup S$, $F_2 \cup S$, can be produced by combining F_1 and F_2 with all successful test cases S separately, and two sets of spectrum information can be collected by executing them on PUT accordingly.[8] The suspiciousness of statements calculated by Ochiai using these two sets of spectrum information is shown in columns $F_1 \cup S$ and $F_2 \cup S$ in Table 3, respectively. In the ranking list produced against $F_1 \cup S$, the statement s_6 where $Fault_1$ lies in is given the highest suspiciousness, while in the ranking list produced against $F_2 \cup S$, the statement s_9 where $Fault_2$ lies in is given the highest suspiciousness. Surprisingly, each faulty statement appears at the top of the corresponding ranking list. Guided by such fault localization outputs with strong pertinence, a developer (in sequential debugging), or two developers (in parallel debugging), only need(s) to inspect at most three statements (the suspiciousness of s_5 and s_6 calculated against $F_1 \cup S$ is identical) for localizing all two faults. However, at least six statements have to be examined for finding two faults in the confusing ranking list produced without clustering failed test cases.

This motivating example not only highlights the promise of clustering failed test cases but also indicates some key factors in such a process: the risk evaluation formula (***REF***) that produces ranking lists to representing failed test cases, the number of successful test cases paired with one individual failed test case (***NSP1F***), may influence clustering effectiveness. Furthermore, considering that the effect of the number of faults (***NOF***) and the fault type (***FT***) in PUT on software debugging has caught the attention of fault localization communities (DiGiuseppe and Jones, 2011b, 2015; Jones et al., 2002), we conjecture these two points are also likely to affect the results of clustering. We conduct extensive controlled experiments to explore how these four factors affect the clustering process in the next section.

Table 5
Subject programs.

Project	Version	kLOC	No. of faults	Description
flex	2.5.3	14.5	30AF + 46PF	Lexical analyzer
grep	2.4	13.5	27AF + 20PF	File patterns searcher
gzip	1.2.2	7.3	24AF + 20PF	Data compressor
sed	3.02	10.2	21AF + 40PF	Text processor
Chart	2.0.0	96.3	18	Chart library
Closure	2.0.0	90.2	36	Closure compiler
Lang	2.0.0	22.1	38	Apache commons-lang
Time	2.0.0	28.4	20	Date and time library

3. Experimental setup

Section 3.1 provides the dataset used in our experiments and the mechanism for generating multi-fault versions via mutation-based strategies. Section 3.2 describes experimental setups for four RQs. Section 3.3 introduces four metrics for evaluating the experimental results.

3.1. The generation of faulty versions

We choose four benchmark programs from SIR (SIR, 2018): *flex*, *grep*, *gzip*, and *sed*, and five benchmark programs from Defects4J (Just et al., 2014a): *Chart*, *Closure*, *Lang*, *Math*, and *Time*, for the generation of multi-fault versions, as shown in Table 5.

3.1.1. SIR programs

SIR (Software-artifact Infrastructure Repository) contains a series of programs written in C that can be expropriated for the use of fault localization. We employ mutation-based strategies to inject multiple artificial faults into four SIR benchmark programs for generating faulty versions (Papadakis et al., 2019). Research such as Andrews et al. (2005), Do and Rothermel (2006), Liu et al. (2006), Andrews et al. (2006), Pradel and Sen (2018) and Just et al. (2014b) has confirmed that mutation-based faults can simulate real-world faults and provide credible results for experiments in the field of software testing and debugging. The following two fault types are defined to mutate source code, which is exemplified in Fig. 2:

- *Assignment Fault* **(AF)**: Editing a variable's value in the statement, or replacing the operators such as addition, subtraction, multiplication, division, etc. with each other (Fig. 2(a));

[7] For more details about the distance metric, the estimation of the number of clusters and the assignment of initial medoids, and the clustering algorithm, please refer to Gao and Wong (2019).

[8] These two fault-focused TSs' coverage and the corresponding spectrum information are omitted due to limited space.

```
// Assignment fault
    add1_buffer(pending_text, ch*2);

// Original code
    add1_buffer(pending_text, ch);
```

(a) Assignment Fault

```
// Predicate fault
    if (start == line.length)

// Original code
    if (start < line.length)
```

(b) Predicate Fault

Fig. 2. Two fault types.

- *Predicate Fault* **(PF)**: Reversing the *if-else* predicate, or deleting the *else* statement, or modifying the decision condition, and so on. (Fig. 2(b)).

After a mutation-based fault is seeded into a benchmark program, a 1-bug faulty version has been generated. To create an r-bug faulty version, the faults from r individual 1-bug faulty versions are injected into the same program. This method of generating a multi-fault version by synthesizing multiple 1-bug faulty versions has been adopted by many studies (Lamraoui and Nakajima, 2016; Yu et al., 2015; Huang et al., 2013).

A total of 960 multi-fault versions have been generated using 228 faults on SIR programs.[9] From the perspective of NOF, they can be categorized into four classes, i.e., 2-bug, 3-bug, 4-bug, and 5-bug, according to how many faults a faulty version contains. On the other hand, from the perspective of FT, they can be categorized into three classes, i.e., TypeA, TypeP, and TypeH, according to the fault type(s) involved in a faulty version.

- **TypeA**: This type of multi-fault version is generated by r 1-bug faulty versions that contain assignment fault (each of r faults contained in a TypeA faulty version is AF).
- **TypeP**: This type of multi-fault version is generated by r 1-bug faulty versions that contain predicate fault (each of r faults contained in a TypeP faulty version is PF).
- **TypeH**: This type of multi-fault version is generated by r 1-bug faulty versions that contain both assignment fault and predicate fault (AF and PF are hybridly contained in a TypeH faulty version).

3.1.2. Defects4J programs

Defects4J gathers a collection of real-world bugs from some open-source projects, due to the realism and ease-to-use, it has been becoming one of the most popular benchmarks in the current field of fault localization. Nonetheless, Defects4J is often utilized in single-fault rather than multi-fault environments, because each of its faulty versions only targets a specific fault. Recently, researchers revisited this benchmark and concluded a new point, that is, many of Defects4J faulty versions actually contain more than one fault, but only one of them can be revealed by the provided test suite. To adapt Defects4J to multi-fault scenarios, An et al. transplanted the fault-revealing test case(s) of another faulty version or other faulty versions to a basic faulty version, that is, enabling a strengthened test suite to detect more faults in the original program (i.e., the basic faulty version) (An et al., 2021).

[9] When two or more specific faults exist in a program, the program may fail to compile, enter an infinite loop, or run for an excessive amount of time. These faulty versions were removed.

Following this strategy, a total of 100 multi-fault versions have been generated using 141 faults on Defects4J programs. It should be highlighted that the generation of multi-fault Defects4J programs involves two limitations. First, it is more difficult to generate multi-fault versions that contain more bugs. The faults in Defects4J come from real-world programming practice, to preserve such a characteristic, we use test cases transplantation instead of source code modification during the generation of multi-fault versions. Specifically, the majority of Defects4J faulty versions are indexed chronologically according to the revision date, a lower ID indicates a more recent version (An et al., 2021), thus the fault in a newer version is also likely to be contained in an older version. For example, we find that the fault in Lang-27 also appears in Lang-28, thus we can add the failed test case of Lang-27 to the test suite of Lang-28, for the generation of a 2-bug version, Lang-27-28. However, it is more difficult to search for a 5-bug version than a 2-bug version, since the more faults, the less likely they co-exist in a same program originally. For this reason, in the created 100 Defects4J multi-fault versions, half of them are 2-bug, and 25, 16, and 9 ones are 3-bug, 4-bug, and 5-bug, respectively. Second, as mentioned above, the faults in Defects4J are not obtained by artificial simulation, thus they cannot be properly categorized into assignment fault or predicate fault. As the consequence of these two problems, Defects4J programs are not suitable for exploring RQ2 (How NOF affects clustering effectiveness?) and RQ3 (Is clustering effectiveness affected by FT?).

In summary, RQ1 and RQ4 will be investigated on all faulty versions that comprise both SIR and Defects4J, considering that these two topics do not involve the number of faults and fault types. And RQ2 and RQ3 will be investigated on SIR, since we can hardly set a proper and fair environment to explore the two questions on Defects4J.

3.2. Experiment setup

In this section, we elaborate on the experimental setups of the four RQs defined in Section 1.

3.2.1. The risk evaluation formulas in SRR (RQ1)

Countless research has been conducted to investigate various REFs in the last four decades (Wong et al., 2016; de Souza et al., 2016). However, most of these studies proposed a novel REF or contrasted existing REFs empirically or theoretically in terms of its/their fault localization effectiveness, that is, analyzing the REF's capability to ranking the faulty statement(s) at the top of the list (Naish et al., 2011; Xie et al., 2013a; Yoo et al., 2017).

For example, some novel REFs have emerged in the past ten years, including Crosstab (Wong et al., 2011) and DStar (Wong et al., 2013) that were developed by Wong et al. in 2011 and 2013, respectively. The former constructs a crosstab for each statement in PUT to determine their suspiciousness by calculating the chi-square statistic and the coefficient of contingency, while the latter exponentially strengthens the function of N_{CF} in spectrum information, making it more effective in fault localization than any other techniques compared with it according to the authors. Yoo created 30 novel REFs via genetic programming in 2012 (Yoo, 2012), experimental results proved that GP-evolved REFs can consistently outperform many of the human-designed REFs. Xie et al. evaluated these 30 GP-evolved REFs using the theoretical framework in Xie et al. (2013a) and discovered three REFs with strong human competitiveness: GP02, GP03, and GP19 (Xie et al., 2013b).

Apart from developing new REFs, some researchers have dedicated their effort to investigating a corpus of existing REFs. For example, Naish et al. investigated more than 30 REFs and

Table 6
35 risk evaluation formulas.

Name	Formula expression	Name	Formula expression
Naish1 (Naish et al., 2011)	$\begin{cases} -1 & \text{if } N_{CF} < N_F \\ N_S - N_{CS} & \text{if } N_{CF} = N_F \end{cases}$	Naish2 (Naish et al., 2011)	$N_{CF} - \frac{N_{CS}}{N_S+1}$
Jaccard (Chen et al., 2002)	$\frac{N_{CF}}{N_F+N_{CS}}$	Anderberg (Naish et al., 2011)	$\frac{N_{CF}}{N_{CF}+2(N_{UF}+N_{CS})}$
Sørensen–Dice (Naish et al., 2011)	$\frac{2N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}$	Dice (Naish et al., 2011)	$\frac{2N_{CF}}{N_F+N_{CS}}$
Goodman (Naish et al., 2011)	$\frac{2N_{CF}-N_{UF}-N_{CS}}{2N_{CF}+N_{UF}+N_{CS}}$	Tarantula (Jones and Harrold, 2005)	$\frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F}+\frac{N_{CS}}{N_S}}$
qe (Lee et al., 2009)	$\frac{N_{CF}}{N_C}$	CBI Inc. (Liblit et al., 2005)	$\frac{N_{CF}}{N_C}$ - $\frac{N_F}{N}$
Wong2 (Wong et al., 2007)	$N_{CF} - N_{CS}$	Hamann (Naish et al., 2011)	$\frac{N_{CF}+N_{US}-N_{UF}-N_{CS}}{N}$
Simple Matching (Naish et al., 2011)	$\frac{N_{CF}+N_{US}}{N}$	Sokal (Naish et al., 2011)	$\frac{2(N_{CF}+N_{US})}{2(N_{CF}+N_{US})+N_{UF}+N_{CS}}$
Rogers & Tanimoto Naish et al. (2011)	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{US}+2(N_{UF}+N_{CS})}$	Hamming etc. (Naish et al., 2011)	$N_{CF} + N_{US}$
Euclid (Naish et al., 2011)	$\sqrt{N_{CF} + N_{US}}$	Wong1 (Wong et al., 2007)	N_{CF}
Russel & Rao (Naish et al., 2011)	$\frac{N_{CF}}{N}$	Binary (Naish et al., 2011)	$\begin{cases} 0 & \text{if } N_{CF}<N_F \\ 1 & \text{if } N_{CF}=N_F \end{cases}$
Scott (Naish et al., 2011)	$\frac{4N_{CF}N_{US}-4N_{UF}N_{CS}-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})(2N_{US}+N_{UF}+N_{CS})}$	Rogot1 (Naish et al., 2011)	$\frac{1}{2}(\frac{N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}+\frac{N_{US}}{2N_{US}+N_{UF}+N_{CS}})$
Kulczynski2 (Naish et al., 2011)	$\frac{1}{2}(\frac{N_{CF}}{N_F} + \frac{N_{CF}}{N_C})$	Ochiai (Abreu et al., 2006)	$\frac{N_{CF}}{\sqrt{N_F N_C}}$
M2 (Naish et al., 2011)	$\frac{N_{CF}}{N_{CF}+N_{US}+2(N_{UF}+N_{CS})}$	Ample2 (Naish et al., 2011)	$\frac{N_{CF}}{N_F} - \frac{N_{CS}}{N_S}$
Wong3 (Wong et al., 2007)	$N_{CF} - h$, where $h=\begin{cases} N_{CS} & \text{if } N_{CS}\le 2 \\ 2+0.1(N_{CS}-2) & \text{if } 2<N_{CS}\le 10 \\ 2.8+0.001(N_{CS}-10) & \text{if } N_{CS}>10 \end{cases}$	Arithmetic mean (Naish et al., 2011)	$\frac{2N_{CF}N_{US}-2N_{UF}N_{CS}}{N_C N_U+N_F N_S}$
Cohen (Naish et al., 2011)	$\frac{2N_{CF}N_{US}-2N_{UF}N_{CS}}{N_C N_S+N_F N_U}$	Fleiss (Naish et al., 2011)	$\frac{4N_{CF}N_{US}-4N_{UF}N_{CS}-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})+(2N_{US}+N_{UF}+N_{CS})}$
Crosstab (Wong et al., 2011)[a]	$\chi^2 = \frac{(N_{CF}-E_{CF})^2}{E_{CF}} + \frac{(N_{CS}-E_{CS})^2}{E_{CS}} + \frac{(N_{UF}-E_{UF})^2}{E_{UF}} + \frac{(N_{US}-E_{US})^2}{E_{US}}$	DStar (Wong et al., 2013)[b]	$\frac{N_{CF}^*}{N_{UF}+N_{CS}}$
GP02 (Yoo, 2012)	$2(N_{CF} + \sqrt{N_{US}}) + \sqrt{N_{CS}}$	GP03 (Yoo, 2012)	$\sqrt{\lvert N_{CF}^2 - \sqrt{N_{CS}}\rvert}$
GP19 (Yoo, 2012)	$N_{CF}\sqrt{\lvert N_{CS} - N_{CF} + N_{UF} - N_{US}\rvert}$		

[a]Crosstab will first calculate φ for each statement to quantify its association with failed and successful executions, and then use φ to determine if a statement should be assigned χ^2, $-\chi^2$ or 0. Please refer to Wong et al. (2011) for more details about this REF.
[b]Considering the preference for DStar in many other studies (such as Pearson et al., 2017; Arrieta et al., 2018), we set $^* = 2$, the most thoroughly-explored value in our experiments.

extracted several equivalence relations guided by the strictest equivalence definition (i.e., only REFs that generate the same statement ranking lists are considered equivalent) (Naish et al., 2011). Xie et al. first excluded some REFs that are not intuitively justified in the context of SBFL, then selected 30 REFs from Naish et al.'s research to contrast them using a novel theoretical framework (Xie et al., 2013a). According to Naish et al. and Xie et al.'s conclusions, 30 REFs are divided into six equivalent groups that include 22 REFs and eight individual REFs.

To the best of our knowledge, no empirical study has been published to investigate how different REFs, which produce ranking lists that represent failed test cases, affect the clustering effectiveness in SRR-based parallel debugging. To fill this gap, we perform the first empirical study on the capability of 35 REFs in Table 6 to representing failed test cases.

3.2.2. The number of faults in PUT (RQ2)

The effect of the number of faults contained in a program on fault localization effectiveness has been investigated by many prior researchers (DiGiuseppe and Jones, 2011b, 2015; Jones et al., 2002), but how NOF affects the clustering stage in parallel debugging is still poorly explored. Although it is intuitive to assume that more bugs will lead to more failures, making it more difficult to divide them, we do not know whether this is reasonable from an empirical standpoint. To that purpose, we observe and compare the effectiveness of clustering in 2-bug, 3-bug, 4-bug, and 5-bug scenarios.

3.2.3. The fault type in PUT (RQ3)

Programmers may introduce various types of faults when coding due to unintentional mistakes or misunderstandings of programming logistics, as a result, FT is typically unpredictable because of the randomness and uncertainty of onsite programming. Lamraoui and Nakajima categorized common faults in multi-fault scenarios into several types, including data-flow dependent faults and control-dependent faults (Lamraoui and Nakajima, 2016). Similar to these, we define assignment faults and predicate faults, two types of faults that are most likely to occur in programming as our research objects, and accordingly generate a series of TypeA faulty versions with only assignment faults, TypeP faulty versions with only predicate faults, and TypeH faulty versions with both two types of faults to observe clustering effectiveness.

3.2.4. The number of successful test cases paired with one individual failed test case (RQ4)

While clustering failed test cases via SRR, many prior studies paired one failed test case with all successful test cases and input them into an REF to produce a ranking list representing this failed test case, without explaining why *all* successful test cases are employed here. In fact, many studies including (Sun et al., 2016; Mottaghi and Keyvanpour, 2017) have managed to utilize test case selection or test suite reduction techniques to lower debugging expenses, some recent studies have also investigated the impact of test suites on fault localization (Lei et al., 2018; Perez et al., 2017). For example, as Fu et al. argued, if the number

Table 7
Four scenarios in the pair of cases-based metric.

Notation	Results of failure indexing	
	In the generated cluster	In the oracle cluster
SS	Same	Same
SD	Same	Difference
DS	Difference	Same
DD	Difference	Difference

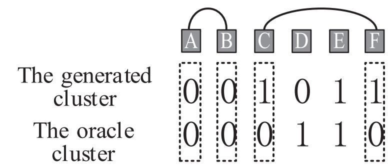


Fig. 3. SS pairs in the generated cluster and the oracle cluster.

Table 8
Four scenarios in the single case-based metric.

Notation	Results of failure indexing	
	In the generated cluster	In the oracle cluster
TP	Positive	Positive
FP	Positive	Negative
TN	Negative	Negative
FN	Negative	Positive

of successful test cases is too large, the noise will be introduced into the fault localization process (Fu et al., 2017). However, these works only evaluated the effect of the number of test cases on fault localization, not fault isolation built upon SRR. We try to cut the scale of successful test cases utilized in SRR by pairing 100%, 80%, 60%, 40%, and 20% of successful test cases with one failed test case, respectively, to monitor if the clustering effectiveness declines as NSP1F falls.

3.3. Metrics

Two classes of metrics, external metrics (Wu et al., 2009) and internal metrics (Tan et al., 2016), are typically implemented to measure the effectiveness of clustering techniques. The former contrast clustering results with the oracle, while the latter examine inherent properties of clustering results, such as compactness and separation, without using an off-the-shelf baseline (Xie et al., 2017). While clustering failed test cases in parallel debugging, ideal outputs should exhibit linkages between each failed test case in TS and each fault in PUT, which is available in our controlled experiments. Therefore, we employ four widely-used external metrics, JC, FMI, PR and RR, to evaluate the experimental results.

3.3.1. Pair of cases-based metric

The pair of cases-based metric refers to compare the indexing consistency of each pair of failed test cases in the generated cluster with the oracle cluster. Four scenarios in which are depicted in Table 7.

Assuming that there are n failed test cases that need to be clustered, a total of C_n^2 pairs will be examined in the pair of cases-based metric. The numbers of pairs that fall into SS, SD, DS, and DD categories are denoted as X_{SS}, X_{SD}, X_{DS}, and X_{DD}, respectively.

The above notations can be incorporated into the Jaccard Coefficient (JC) and the Fowlkes and Mallows Index (FMI), which are defined in Formula (4) and Formula (5), respectively. JC and FMI are used to determine the similarity between the generated cluster and the oracle cluster, for measuring the clustering results (Huang and Flynt, 2018).

$$JC = \frac{X_{SS}}{X_{SS} + X_{SD} + X_{DS}} \tag{4}$$

$$FMI = \sqrt{\frac{X_{SS}}{X_{SS} + X_{SD}} \times \frac{X_{SS}}{X_{SS} + X_{DS}}} \tag{5}$$

It can be proved that the intervals of JC and FMI are both [0, 1], and that the larger the value in this range, the more effective clustering is. A simple example is given below to describe JC and FMI.

As shown in Fig. 3, six failed test cases (A, B, C, D, E, and F) are indexed divergently in the generated cluster and the oracle cluster. Among the $C_6^2 = 15$ pairs of cases ($A-B$, $A-C$, $A-D$, ..., $E-F$), $A-B$ and $C-F$ are in the *same* cluster in the generated cluster, and also in the *same* cluster in the oracle cluster, which meets the scenario SS in Table 7, therefore, $X_{SS} = 2$. Similarly, we can get $X_{SD} = 4$, $X_{DS} = 5$, and $X_{DD} = 4$. Incorporating these notations into Formulas (4) and Formula (5), JC and FMI will be set to 0.182 and 0.309, respectively.

3.3.2. Single case-based metric

The single case-based metric refers to compare the classification result of each failed test case in the generated cluster with the oracle cluster. Four scenarios in which are depicted in Table 8.

The numbers of failed test cases that fall into TP, FP, TN, and FN categories are denoted as X_{TP}, X_{FP}, X_{TN}, and X_{FN}, respectively.

The above notations can be incorporated into the Precision Rate (PR) and the Recall Rate (RR), which are defined in Formula (6) and Formula (7), respectively, for measuring the clustering results.

$$PR = \frac{X_{TP}}{X_{TP} + X_{FP}} \tag{6}$$

$$RR = \frac{X_{TP}}{X_{TP} + X_{FN}} \tag{7}$$

It can be proved that the intervals of PR and RR are both [0, 1], and that the larger the value in this range, the more effective clustering is.

As shown in Fig. 3, failed test cases D and E are labeled as positive, and the remaining four ones are labeled as negative in the oracle cluster. But in the generated cluster, failed test cases C and F are wrongly labeled as positive, thus the value of X_{FP} can be determined as 2. Similarly, we can get $X_{TP} = 1$, $X_{TN} = 2$, and $X_{FN} = 1$. Incorporating these notations into Formulas (6) and Formula (7), PR and RR will be set to 0.333 and 0.5, respectively.

3.3.3. The virtual mapping problem

It should be noted that the different permutations between generated clusters and oracle clusters will result in different outputs of the external metrics, and the diversity of permutations will significantly grow with the number of faults increases. For example, in a 2-bug scenario, the permutations between two generated clusters and two oracle clusters are $A_2^2 = 2$, while in a 5-bug scenario, the permutations between five generated clusters and five oracle clusters are $A_5^5 = 120$. Such diversity of permutations does not exist in practical parallel debugging, it only occurs in the contrast between the output and the oracle. In other words, each developer will be allocated to a fault-focused TS and will be responsible for localizing the corresponding fault independently, thus regardless of how many potential permutations exist, there is **only one** real combination of generated clusters and oracle clusters. As a result, the *permutation* between generated clusters and oracle clusters in experiments is the *combination* in practice (we call this problem the *virtual mapping problem*). In our experiments, we extract faulty versions in which the number of faults has been precisely estimated, i.e., the number of faults equals the number of generated clusters, to perform analyses. For each

Table 9
12 groups of risk evaluation formulas with the same capability to representing failed test cases.

Name	REFs
Group1	**Naish2**
Group2	**Jaccard**, Anderberg, Sørensen–Dice, Dice, Goodman, M2, Naish1, DStar
Group3	**Tarantula**, qe, CBI Inc, Kulczynski2, Ochiai
Group4	**Wong2**, Hamann, Simple Matching, Sokal, Rogers & Tanimoto, Hamming etc., Euclid
Group5	**Wong1**, Binary, Russel & Rao
Group6	**Scott**, Rogot1
Group7	**Ample2**, Arithmetic Mean, Cohen, Crosstab
Group8	**Wong3**
Group9	**Fleiss**
Group10	**GP02**
Group11	**GP03**
Group12	**GP19**

of these faulty versions, we enumerate all feasible permutations followed by picking the optimal one based on the value of JC, FMI, PR, or RR for evaluation, because which permutation reflects the real mapping relations is unknown.

4. Result and analysis

We conduct extensive controlled experiments according to the research questions in Section 1 and predesigned setups in Section 3. Experimental results and analyses are given in this section.

4.1. The capability of different REFs to representing failed test cases (RQ1)

We reorganize 35 REFs in Table 6 into 12 disjoint groups, as shown in Table 9, because we find that some REFs have the same performance in representing failed test cases (details are omitted to conserve space). Only one REF (in bold) in each group is selected for analyses since its capability to representing failed test cases is equal to the others in the group it belongs to.

For each of faulty versions, we implement the workflow shown in Fig. 1 to estimate the number of clusters based on the ranking lists produced by an REF. There are three scenarios in this stage:

- **Under:** The estimated number of clusters is fewer than NOF (i.e., $k < r$).
- **Equal:** The estimated number of clusters is equal to NOF (i.e., $k == r$).
- **Over:** The estimated number of clusters exceeds NOF (i.e., $k > r$).

If the estimated number of clusters k in a faulty version is equal to NOF r, we send this faulty version to the next clustering step. Otherwise, if k is not equal to r, this faulty version is discarded. In a real multi-fault localization scenario, even if the estimated number of clusters k and the number of faults r are not identical, the whole process can also be continued: if $k > r$, localization can be stopped when all failures disappear, and if $k < r$, localization can be carried out more than one iteration. This paper focuses on clustering rather than the following localization stage, the evaluation of clustering effectiveness is the main purpose, thus we do not take "$k \neq r$" scenarios into account. The filtering, as well as the follow-up virtual mapping process, are illustrated in Fig. 4.

When estimating the number of clusters based on the ranking lists produced by an REF R, we denote the numbers of faulty versions that fall into the *Under*, *Equal*, and *Over* categories as V^R_{under}, V^R_{equal}, and V^R_{over}, respectively. For an REF R, a greater V^R_{equal}, as well as a fewer V^R_{under} and a fewer V^R_{over}, partly indicate that R captures the execution features of failed test cases more effectively thus

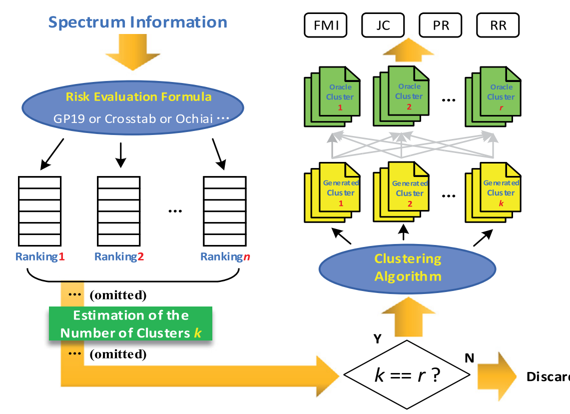


Fig. 4. The virtual mapping process (with checking whether the estimated number of clusters equals NOF).

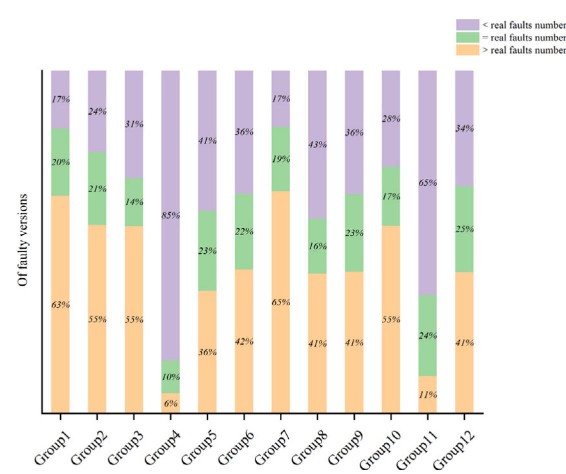


Fig. 5. V^R_{under}, V^R_{equal}, and V^R_{over} of 12 groups of REFs.

can better represent them. V^R_{under}, V^R_{equal}, and V^R_{over} of each group of REFs on all faulty versions are shown in Fig. 5.[10]

It can be seen that based on the ranking lists produced by Group12, NOF is accurately estimated on 25% of the 1060 faulty versions ($V^{Group12}_{equal} = 265$). Besides, based on the ranking lists produced by Group7, the estimated numbers of clusters on 65% of faulty versions exceed the NOF ($V^{Group7}_{over} = 687$), implying this REF is *over-representing* in modeling failed test cases (i.e., too

10 The longer the *green* band, the more faulty versions' NOF can be accurately estimated based on the ranking lists produced by the corresponding REF.

Table 10
Contrast of the capability of 12 groups of REFs to representing failed test cases.

Versus R_1 / R_2	Group1	Group2	Group3	Group4	Group5	Group6	Group7	Group8	Group9	Group10	Group11	Group12
Group1		✓✓✓✓			✓✓✓✓	✓✓✓✓			✓✓✓✓		✓✓✓✓	✓✓✓✓
Group2					✓✓✓✓	✓✓✓✓			✓✓✓✓		✓✓✓✓	✓✓✓✓
Group3	✓✓✓✓	✓✓✓✓			✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Group4	✓✓✓✓	✓✓✓✓	✓✓✓✓		✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Group5									✓		✓ ✓✓	✓✓✓✓
Group6					✓✓✓✓				✓✓✓✓		✓✓✓✓	✓✓✓✓
Group7	✓✓✓✓	✓✓✓✓			✓✓✓✓	✓✓✓✓			✓✓✓✓		✓✓✓✓	✓✓✓✓
Group8	✓✓✓✓	✓✓✓✓	✓		✓✓✓✓	✓✓✓✓	✓✓✓✓		✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓
Group9					✓✓ ✓						✓✓✓✓	✓✓✓✓
Group10	✓✓✓✓	✓✓✓✓			✓✓✓✓	✓✓✓✓	✓✓✓✓		✓✓✓✓		✓✓✓✓	✓✓✓✓
Group11					✓							✓✓✓✓
Group12												

Table 11
The values of *Sum_Metric* of 12 groups of REFs.

$Sum\_Metric^R_M$ M / R	FMI	JC	PR	RR
Group1	173.71	153.84	174.06	131.56
Group2	187.3	166.28	190.2	144.81
Group3	128.32	116.7	122.89	108.87
Group4	90.11	82.65	89.7	78.22
Group5	205.81	182.04	194.5	157.79
Group6	193.78	170.71	192.12	151.15
Group7	164.79	146.17	164.8	125.25
Group8	140.31	123.99	135.45	107.17
Group9	196.54	172.78	194.8	152.4
Group10	150.27	133.76	147.51	117.59
Group11	206.48	181.21	205.38	171.01
Group12	**221.33**	**196.62**	**227.15**	**177.47**

sensitive to model failures to a nicety). Based on the ranking lists produced by Group4, the estimated numbers of clusters in 85% of faulty versions are fewer than the NOF ($V^{Group4}_{under} = 896$), indicating that this REF appears *under-representing* in modeling failed test cases (i.e., too deficient to model failures distinguishably).

We select only faulty versions that fall into the *Equal* category (i.e., satisfy the "$k == r$" criteria) for clustering, as illustrated in Fig. 4. The capability of an REF R to representing failed test cases can be assessed by two indicators: the value of V^R_{equal} and the clustering effectiveness on these V^R_{equal} faulty versions. We define the $Sum\_Metric^R_M$, as shown in Formula (8), to incorporate the two metrics into a single value.

$$Sum\_Metric^R_M = \sum_{i}^{V^R_{equal}} M_i \tag{8}$$

where R represents the REF, M_i is the value of the clustering metric M (M takes FMI, JC, PR or RR) on the ith faulty version. For instance, if we want to evaluate Group12 from the standpoint of FMI (i.e., R takes Group12 and M takes FMI) using Formula (8), we can first get the value of $V^{Group12}_{equal}$ (265), and then calculate $Sum\_Metric^{Group12}_{FMI}$ by adding up $FMI_1, FMI_2, \ldots, FMI_{265}$. Specifically, the values of FMI on the first, the second, …, and the 265$^{\text{th}}$ version are 0.78, 0.77, …, and 1.00, respectively, thus the value of $Sum\_Metric^{Group12}_{FMI}$ can be determined by adding up 0.78, 0.77, …, and 1.00, that is, 221.33 in Table 11. Obviously, the greater the V^R_{equal} value of the REF R, the more possibility it has to obtain a greater $Sum\_Metric^R_M$.

For two REFs, R_1 and R_2, if $Sum\_Metric^{R_1}_M > Sum\_Metric^{R_2}_M$, it means that according to the metric M, R_1 is better than R_2 in representing failed test cases. We contrast 12 groups of REFs according to their $Sum\_Metric^R_M$ in Table 10,[11] as well as list the $Sum\_Metric^R_M$ values in Table 11. It can be seen that Group 12 outperforms the other 11 groups of REFs regardless of being evaluated by FMI, JC, PR, or RR.

Now we can draw the conclusion of RQ1: Group12 is highly competitive across all REFs when representing failed test cases. The list of 12 groups of REFs ranked by their capability to representing failed test cases is as follows:

> **Group12** > Group11 > Group5 > Group9 > Group6 > Group2 > Group1 > Group7 > Group10 > Group8 > Group3 > Group4

4.2. The impact of NOF contained in PUT on the clustering effectiveness (RQ2)

Similar to definitions depicted in Section 4.1, we first use V^N_{equal} ($N = 2, 3, 4, 5$) to denote how many N-bug faulty versions' NOF can be accurately estimated by a specific REF, then define and employ the $Sum\_Metric^N_M$, as illustrated in Formula (9), to observe the clustering effectiveness on these V^N_{equal} versions.

$$Sum\_Metric^N_M = \sum_{i}^{V^N_{equal}} M_i \tag{9}$$

The clustering effectiveness is visualized using box-and-whisker plots in terms of upper quartile, lower quartile, median, and mean, where each vertical column's color reflects the value of V^N_{equal} (a darker color indicates a greater value of V^N_{equal}). The color is regulated by adjusting the opacity using the procedures below.[12]

- **Step-1**: Set the color of each vertical column to *black* (RGB: 0, 0, 0).
- **Step-2**: Count the values of V^N_{equal} in N-bug scenarios ($N = 2, 3, 4, 5$), and set the maximum value to MAX, as defined in Formula (10).

$$MAX = \max\left\{V^N_{equal}\right\} \quad (N = 2, 3, 4, 5) \tag{10}$$

- **Step 3**: Calculate the opacity $Opacity_N$ of each vertical column, as defined in Formula (11).

$$Opacity_N = \frac{V^N_{equal}}{MAX} \quad (N = 2, 3, 4, 5) \tag{11}$$

11 In the cell of [R_1, R_2], ✓, ✓, ✓, ✓ indicate that REF R_1 is better than REF R_2 in terms of FMI, JC, PR, or RR, respectively.

12 This color setting scheme is also applicable to Section 4.3.

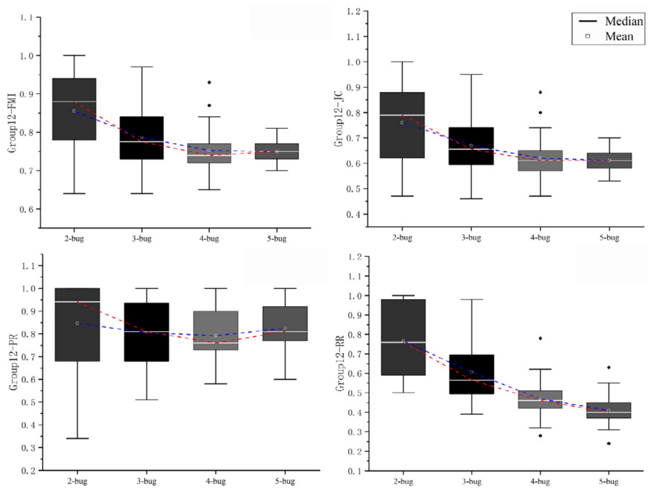


Fig. 6. The contrast of clustering effectiveness among 2-bug, 3-bug, 4-bug, and 5-bug scenarios.

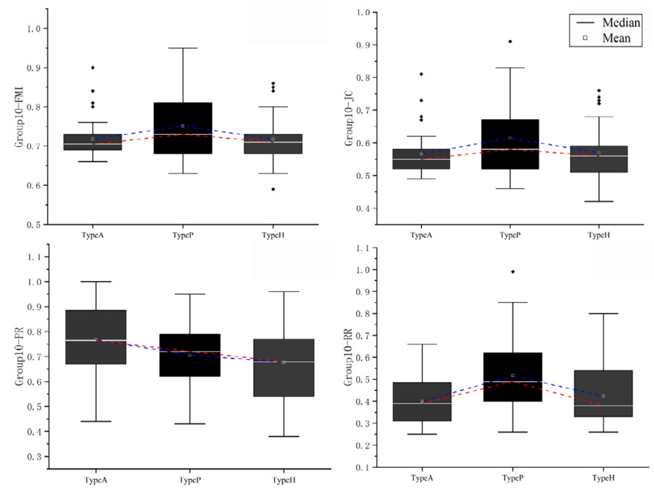


Fig. 7. The contrast of clustering effectiveness among TypeA, TypeP, and TypeH scenarios.

The clustering effectiveness in 2-bug, 3-bug, 4-bug, and 5-bug scenarios is shown in Fig. 6.[13] From this, **we can draw the conclusions of RQ2:**

(1) As NOF increases, the similarities (FMI, JC) between generated clusters and oracle clusters decrease.
(2) As NOF increases, the Recall Rate (RR) falls, while the Precision Rate (PR) changes little.
(3) As NOF increases, the dispersion of FMI, JC, and RR narrows.
(4) Based on the ranking lists produced by Group12, a greater value of V^{N}_{equal} tends to be obtained if N equals 3 (Group3, Group5, Group8, and Group10 also support this conclusion).

The list of NOFs ranked by the clustering effectiveness under them is as follows:

2-bug > 3-bug > 4-bug > 5-bug

4.3. The impact of FT contained in PUT on the clustering effectiveness (RQ3)

Similar to definitions depicted in Section 4.1, we first use V^{T}_{equal} (T takes A, P, H) to denote how many Type$T$ faulty versions' NOF can be accurately estimated by a specific REF, then define and employ the $Sum\_Metric^{T}_{M}$, as illustrated in Formula (12), to observe the clustering effectiveness on these V^{T}_{equal} versions.

$$Sum\_Metric^{T}_{M} = \sum_{i}^{V^{T}_{equal}} M_i \tag{12}$$

The clustering effectiveness in TypeA, TypeP, and TypeH scenarios is shown in Fig. 7.[14] From this, **we can draw the conclusions of RQ3:**

(1) Compared with TypeA and TypeH, better clustering effectiveness is easier to obtain in the TypeP scenario concerning FMI, JC, and RR. No significant differences in terms of PR among the three scenarios are observed.
(2) The values of V^{T}_{equal} and T have no evident relations.

The list of FTs ranked by the clustering effectiveness under them is as follows:

TypeP > TypeA ≈ TypeH

4.4. The impact of NSP1F on the clustering effectiveness (RQ4)

Unlike the first three RQs in which we pair one failed test case with all (i.e., 100%) successful test cases, we randomly sample X% ($X = 80, 60, 40, 20$) of successful test cases to pair with one failed test case in this RQ. Similar to definitions depicted in Section 4.1, we first use V^{X}_{equal} to denote how many faulty versions' NOF can be accurately estimated by a specific REF when the proportion of successful test cases is set to X%, then define and employ the $Sum\_Metric^{X}_{M}$, as illustrated in Formula (13), to observe the clustering effectiveness on these V^{X}_{equal} versions.

$$Sum\_Metric^{X}_{M} = \sum_{i}^{V^{X}_{equal}} M_i \tag{13}$$

The clustering effectiveness when X is set to 100, 80, 60, 40, 20 is shown in Table 12.[15] For example, "FMI-mean-80%: 0.82" implies that when pairing one failed test case with 80% of successful test cases, the mean of the values of *FMI* on 255 "$k == r$" faulty versions is 0.82. From this, **we can draw the conclusions of RQ4:**

(1) Lowering NSP1F (to as low as 20%) has no evident effect on clustering effectiveness.
(2) The effect of X on the value of V^{X}_{equal} is neither evident nor decisive.

[13] Due to space limitations, we only display the clustering results of Group12, despite the fact that the clustering results of the other 11 groups of REFs all confirm the conclusions in Section 4.2. Please refer to the supplementary material for a complete list of conclusions.

[14] Due to space limitations, we only display the clustering results of Group10, despite the fact that the clustering results of many of the other 11 groups of REFs confirm the conclusions in Section 4.3. Please refer to the supplementary material for a complete list of conclusions.

[15] Due to space limitations, we only display the clustering results of Group11, despite the fact that the clustering results of the other 11 groups of REFs all confirm the conclusions in Section 4.4. Please refer to the supplementary material for a complete list of conclusions.

Table 12
The contrast of clustering effectiveness among various NSP1Fs.

M \ the values of M \ X		100%	80%	60%	40%	20%
FMI	mean	0.82	0.82	0.82	0.81	0.81
	median	0.79	0.79	0.79	0.79	0.79
JC	mean	0.72	0.71	0.71	0.71	0.70
	median	0.67	0.66	0.67	0.66	0.66
PR	mean	0.82	0.81	0.81	0.81	0.80
	median	0.85	0.82	0.80	0.81	0.79
RR	mean	0.68	0.67	0.66	0.67	0.65
	median	0.64	0.61	0.59	0.60	0.57
	V^{X}_{equal}	**251**	**255**	**261**	**261**	**263**

The list of NSP1Fs ranked by clustering effectiveness under them is as follows:

100% ≈ 80% ≈ 60% ≈ 40% ≈ **20%**

This conclusion indicates that 100% clustering effectiveness can be achieved with only 20% of successful test cases. When developers use SRR to clustering failed test cases in parallel debugging, they can feel free to cut the scale of successful test cases for lower debugging costs without worrying about the loss of effectiveness.

5. Discussion

Some interesting topics related to our empirical study are further discussed in this section.

5.1. An in-depth analysis of clustering failed test cases

Given a TS and a PUT, the numbers of failed test cases and successful test cases will be immediately determined. If multiple faults are contained in the PUT, all existing failed test cases might be caused by different faults, that is, each failed test case will be linked to its root cause(s). The more the faults, the lower proportion of failed test cases caused by each fault to all failed test cases.[16] However, the intuition of designing risk evaluation formulas in SBFL is to assign higher suspiciousness to statements that are covered by more failed test cases (Tang et al., 2017; Pang et al., 2015), which would be disturbed by the presence of multiple faults, and the degree of disturbance magnifies as the number of faults increases. Zheng et al. presented a similar opinion in Zheng et al. (2018), they claimed when there is only one faulty statement, it is more likely to be covered by more failing executions, whereas the failing executions are diluted by multiple faults so less accurate results are obtained.

To tackle this challenge, it is natural to categorize failed test cases according to their root cause(s), in other words, build linkages between failed test cases and faults. As a classic technique for unsupervised data grouping, clustering is typically employed to accomplish this failure indexing process, with the goal of fault isolation.

We use Fig. 8 to simulate the effectiveness of fault isolation. In a single-fault scenario, the proportion of failed test cases caused by the unique fault F_1 (denoted as valid failed test cases for F_1) to all failed test cases is 100%, that is to say, all failed test cases fed into a risk evaluation formula for F_1, thus SBFL techniques are

16 We discuss this problem under the condition of the number of failed test cases has been determined.

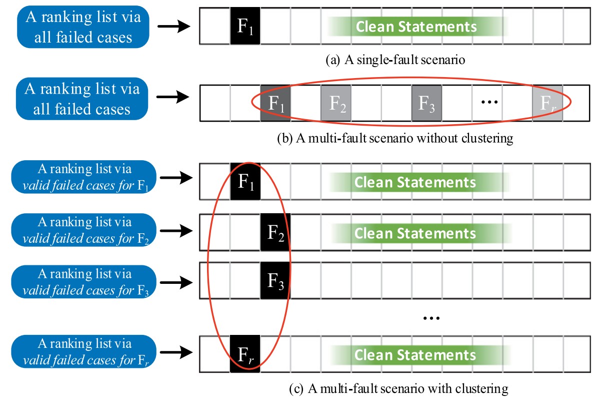


Fig. 8. Fault localization effectiveness with and without clustering in a multi-fault scenario.

easier to push the statement that contains F_1 towards the top of the ranking list, as shown in Fig. 8(a). In a multi-fault scenario, assume there are r bugs, F_i ($i = 1, 2, \dots, r$) in a PUT, n failed test cases in a TS, and the number of failed test cases caused by F_i (denoted as valid failed test cases for F_i) is $|F_i|$. The proportion of failed test cases linked to F_i to all failed test cases is $|F_i|/n$, which is ordinarily less than 100%. Furthermore, if all failed test cases are utilized in SBFL without being refined, the process of localizing a single fault, F_i, will be interrupted by failed test cases caused by the other faults (denoted as redundant failed test cases for F_i). Consequently, SBFL techniques' capability is diminished since linkages between a single fault and its responsible failed test cases have **been diluted** (simulated by the opacity of faulty statements in Fig. 8(b)), potentially lowering the rankings of statements that contain faults.

After all failed test cases are divided into several disjoint fault-focused clusters, only failed test cases triggered by F_i, as well as successful test cases, will be fed into a risk evaluation formula to localize F_i. That is to say, when ideal clustering results are delivered, the proportion of valid failed test cases for F_i to all failed test cases **regains** 100%, since redundant failed test cases for F_i have been indexed to their own root cause, which enables the position of the statement that contains F_i to be higher in the corresponding ranking list, as shown in Fig. 8(c).

5.2. Revisit of V^R_{over} and V^R_{under}

When evaluating the capability of REFs to representing failed test cases, we consider only faulty versions that fall into the *Equal* category, in other words, if the NOF of a faulty version is not accurately estimated based on an REF R (i.e., falls into the *Under* or the *Over* category), this faulty version will be discarded, and thus will not be dedicated to R's capability to clustering failed test cases. It is obvious that the larger the value of V^R_{equal} (that is, the lower the values of V^R_{over} and V^R_{under}), the greater the possibility that R will be highly competitive.

Nonetheless, the same values of V^R_{over} and V^R_{under} should not be treated equally since they can reflect different deviations from the NOF. For example, assume that the NOFs of ten 5-bug faulty versions are being estimated based on the ranking lists produced by two REFs, R_1 and R_2, respectively, we can immediately get r_i ($i = 1, 2, \dots, 10$) are all equal to 5. If the estimate results generated by R_1 are $k^{R_1}_i$ ($i = 1, 2, \dots, 10$), which are 9, 9, 8, 9, 5, 5, 1, 1, 2, 2, respectively, and the estimate results generated by R_2 are $k^{R_2}_i$ ($i = 1, 2, \dots, 10$), which are 6, 6, 7, 6, 5, 5, 3, 3, 4, 4, respectively. According to the preceding definitions in Section 4.1, the values of $V^{R_1}_{over}$ and $V^{R_2}_{over}$ are equal to 4, the values of $V^{R_1}_{equal}$

Table 13
The values of *Deviation* of 12 groups of REF.

Value / Metrics / R	$Deviation^R_{over}$	$Deviation^R_{under}$	*mean*
Group1	3.32	**1.57**	2.95
Group2	2.89	1.70	2.53
Group3	4.81	1.81	3.72
Group4	1.95	2.39	2.36
Group5	2.02	1.73	**1.87**
Group6	2.37	1.85	2.13
Group7	3.46	1.66	3.09
Group8	4.21	2.15	3.15
Group9	2.35	1.87	2.13
Group10	3.80	2.04	3.20
Group11	**1.26**	2.05	1.94
Group12	2.07	1.80	1.95

and $V^{R_2}_{equal}$ are equal to 2, and the values of $V^{R_1}_{under}$ and $V^{R_2}_{under}$ are equal to 4. Although both R_1 and R_2 estimate the NOF on eight faulty versions inaccurately, it is visible that R_2 delivers a closer result, implying R_2 has a stronger capability to representing failed test cases to some extent. We define two metrics, $Deviation^R_{over}$ in Formula (14) and $Deviation^R_{under}$ in Formula (15), to quantify this type of difference among all REFs.

$$Deviation^R_{over} = \frac{1}{V^R_{over}} \times \sum_{i}^{V^R_{over}} (k_i - r_i) \tag{14}$$

$$Deviation^R_{under} = \frac{1}{V^R_{under}} \times \sum_{i}^{V^R_{under}} (r_i - k_i) \tag{15}$$

where k_i is the estimated number of clusters on the ith faulty version, and r_i represents the NOF contained in the ith faulty version.

Using Formula (14) and Formula (15) to contrast R_1 and R_2 in the aforementioned example, we can get $Deviation^{R_1}_{over} = 3.75$, $Deviation^{R_1}_{under} = 3.5$; $Deviation^{R_2}_{over} = 1.25$, $Deviation^{R_2}_{under} = 1.5$. Hence, the difference between R_1 and R_2 hidden behind the $k \neq r$ faulty versions is captured and quantified.

We revisit the values of V^R_{over} and V^R_{under} for 12 groups of REFs presented in Fig. 5, as shown in Table 13.

It can be seen that the value of $Deviation^R_{over}$ of Group11 is 1.26, indicating when the estimated number of clusters exceeds the NOF, Group11 has the lowest degree of *over-representing*. The value of $Deviation^R_{under}$ of Group1 is 1.57, indicating when the estimated number of clusters is fewer than the NOF, Group1 has the lowest degree of *under-representing*. The *mean* of Group5 is 1.87, indicating when the estimated number of clusters is not equal to the NOF, Group5 has the lowest deviation.

Notice that such analyses are non-trivial for parallel debugging. In real multi-fault localization scenarios, it is expected that the predicted number of faults k is identical to the number of faults r. If such ideal situations cannot be attained, the smaller the deviation, the lower the time and labor cost. Specifically, one cannot judge whether the prediction result is correct since the value of r is unknown in practice. Thus, k fault-focused clusters will be directly input to the following localization stage. If k exceeds r, k developers will be employed to locate r faults, resulting in waste of human labor (k - r developers are redundant). On the contrary, if k is less than r, more than one ($\lceil r/k \rceil$) iteration of debugging is needed, resulting in waste of time.

5.3. A heuristic perspective to contrast REFs

We further discuss the relation between the virtual mapping problem and the evaluation of clustering effectiveness. Assume that REF R is utilized to represent failed test cases in a faulty version. If the estimated number of clusters k is equal to the NOF r, there will be A^r_k permutations between generated clusters and oracle clusters. The four metrics, FMI, JC, PR, and RR, will appear different values on different permutations. If the highest values of the four metrics all appear on the same permutation, it means that the four metrics can easily achieve a consensus, which indicates that the ranking lists produced by R represent failed test cases distinguishably. On the contrary, if the highest values of the four metrics are dispersed onto different permutations, divergences among these four metrics are revealed, which just demonstrates that the ranking lists produced by R are too analogous to be divided.

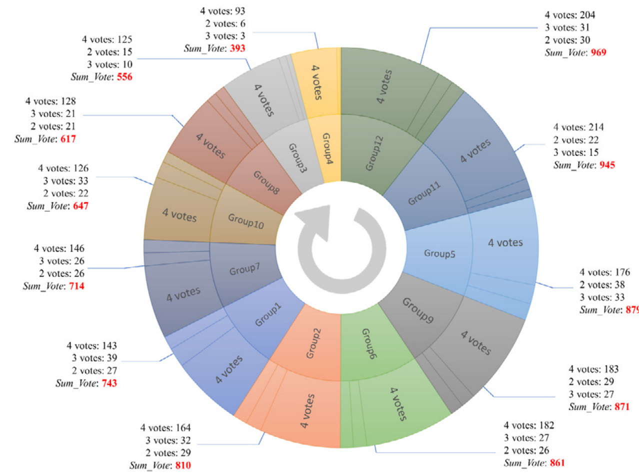


Fig. 9. The values of Sum_Vote^R of 12 groups of REFs.

We regard the evaluation of four metrics for all permutations as a voting process, in which each metric votes for the permutation with its highest value. For example, a permutation will get four votes if the highest values of all four metrics occur on it. Obviously, in the aforementioned r-bug faulty version, A^r_k permutations will each be assigned a value of votes. This r-bug faulty version's votes will be referred to as the highest value of votes among A^r_k permutations.

We design the Sum_Vote^R metric to count the votes of faulty versions that satisfy the "$k == r$" criteria for each REF R in Fig. 5, as shown in Formula (16). We believe that the Sum_Vote^R metric reflects the capability of the risk evaluation formula R to representing failed test cases from a heuristic perspective.

$$Sum\_Vote^R = \sum_{i}^{V^R_{equal}} vote_i \tag{16}$$

where $vote_i$ is the value of votes of the ith faulty version.

The values of Sum_Vote^R of 12 groups of REFs are given in Fig. 9. For instance, on 265 "$k == r$" faulty versions of Group12, 204, 31, and 30 of them get 4, 3, and 2 votes, respectively, we can immediately obtain $Sum\_Vote^{Group12} = 969$ according to Formula (16). The direction of the circular arrow in Fig. 9 indicates the ranking of Sum_Vote^R values of 12 groups of REFs: Group12 > Group11 > Group5 > Group9 > Group6 > Group2 > Group1 > Group7 > Group10 > Group8 > Group3 > Group4, double-confirming the conclusion of RQ1.

5.4. Why is it easier to obtain better clustering effectiveness in TypeP faulty versions?

The conclusions in Section 4.3 reveal that when a program has only predicate faults, the overall clustering effectiveness is higher than when it has only assignment faults and both two types of faults coexist. Take Group10 as an example (Fig. 7), the number of "$k == r$" faulty versions of TypeP is 25.0% and 28.6% greater than that of TypeA and TypeH, respectively, according to their opacity. TypeP scenarios also have better clustering effectiveness (the mean and median of FMI, JC, and RR) than the other two fault types.

In SRR-based failure clustering, a ranking list, which is produced by a risk evaluation formula, serves as a proxy for a failed test case. The basis of generating a ranking list is spectrum information, while the latter originates from coverage. In other words, SRR-based failure clustering heavily depends on the failed test cases' execution paths on the PUT. For failed test cases caused by different faults, the more distinctive execution paths they have, the more distinguishable ranking lists an REF can generate, and the easier they are to be indexed. A TypeP faulty version has only predicate faults, which involve reversing the *if-else* predicate, deleting the *else* statement, or modifying the decision condition, etc., according to the definition in Section 3.1.1. All of the three classes could cause unwanted code to be executed, resulting in a different trace. Thus, failed test cases in TypeP faulty versions are more likely to appear diverse coverage, which will be beneficial to isolate these predicate faults. However, this assistance, on the one hand, does not exist when a program contains only assignment fault, on the other hand, is diminished when the two types of faults coexist.

5.5. The function of successful test cases in SRR

In Section 3.2.4, we assume that the function of successful test cases in SRR-based failure clustering is to assist risk evaluation formulas in generating ranking lists (some REFs will lose their definition without being fed into successful test cases), that is to say, they serve as complements in failure indexing. The conclusions in Section 4.4 reveal that lowering NSP1F (to as low as 20%) indeed has no evident effect on clustering effectiveness. As a result, while performing SRR-based failure clustering, developers can reduce debugging costs by pairing only a portion of the successful test cases with one failed test case since too many successful test cases will not help represent failed test cases.

Even though failed test cases have gotten a lot of attention in testing and debugging, successful test cases can also play a vital role. For example, metamorphic testing enables successful test cases to expose failures via metamorphic relations (Chen et al., 2020; Xie et al., 2013c). We only illustrate the redundancy of successful test cases in SRR-based failure clustering, without denying their significance in localization, testing, or the other software quality assurance activities.

6. Threats to validity

Similar to previous empirical studies on parallel debugging, a hard-clustering strategy is used in this paper to divide failed test cases, that is, a failed test case can only be categorized into one cluster. However, in real-world debugging processes, the relations between faults and failures are quite complex since several faults might trigger the same failure (i.e., one failed test case links to multiple faults). Therefore, the clustering effectiveness will be reduced since the inherent conflict between the property of hard-clustering techniques and the one-to-many or many-to-many linkages. Nonetheless, the reliability of our conclusions is not affected by this threat since we contrast different variables based on the same clustering technique.

In addition, to build the virtual linkages between generated clusters and oracle clusters, we filter out faulty versions with the estimated number of clusters not equal to the NOF. Although this strategy guarantees the availability of clustering results, it also causes various variables in each RQ to be contrasted based on different numbers of faulty versions. This threat seems to introduce additional uncertainties for the experiments, however, we believe that (1) how many faulty versions are selected by various variables in each RQ (i.e., 12 groups of REFs in RQ1, 2-bug, 3-bug, 4-bug, and 5-bug scenarios in RQ2, TypeA, TypeP, and TypeH scenarios in RQ3, 100%, 80%, 60%, 40%, and 20% of successful test cases in RQ4) reflect these variables' capability to representing failed test cases, and (2) the distinction in diverse benchmarks avoids the bias caused by a standard dataset, which makes the conclusions more universal.

Although we collected four datasets with varied scales and functions, they are all written in C. Besides, when utilizing the mutation-based strategy to inject faults into the original program, the number of predefined mutation operators is limited, which lowers the diversity of faulty versions to some extent.

7. Related work

Clustering failed test cases into various fault-focused groups that target different faults is not a newborn method. As early as 2003, Podgurski et al. observed that open-source software developers had received a large number of bug reports from end-users every day, but many of these bug reports are actually caused by the same fault although they have distinct trigger paths and different anomalous behaviors. To that end, they suggested grouping together failures with the same root cause based on supervised and unsupervised pattern classification, which avoids potentially unwanted and redundant debugging labor (Podgurski et al., 2003). Considering the suggestions of Podgurski et al. (2003), Jones et al. proposed two parallel debugging techniques in Jones et al. (2007). Specifically, they first divided failed test cases into several disjoint clusters based on similarities, and then separately combined these clusters with all successful test cases to generate specialized test suites that are expected to target different faults. These fault-focused TSs are finally assigned to several developers for localizing multiple faults in parallel. DiGiuseppe and Jones then conducted an empirical study to confirm the necessity of clustering failed test cases and to explore the influence of the presence of multiple faults on fault localization. They pointed out clustering failed test cases is necessary and beneficial despite the fact that this process may incur additional computational costs, since their findings demonstrated that multi-fault indeed had a negligible effect on the effectiveness of fault localization (DiGiuseppe and Jones, 2011b).

Högerle et al. first quoted an important opinion concluded by Jones et al. in Jones et al. (2007), that is, parallelization can speed up debugging significantly, even if the derived parallel tasks are conducted sequentially, and then pointed out that the method of dividing failed test cases should be carefully chosen because it will have a significant impact on the division effectiveness through large-scale experiments (Högerle et al., 2014). The effectiveness of parallel debugging will be directly determined by the outcomes of the clustering process. Zakari and Lee investigated commonly-used parallel debugging techniques and found that most research (1) employed CVR as failure proximity to represent failed test cases, and (2) used Euclidean, Jaccard, or Hamming distance to measure the similarities between failed test cases. They first coined the term *problematic approach* to describe debugging approaches that adopted the above techniques, and then conducted an empirical study on the effectiveness of several problematic approaches adopting the K-means clustering algorithm. Their results showed that clustering built upon CVR and Euclidean distance reduced the effectiveness of multi-fault localization (Zakari and Lee, 2019).

Liu et al. conducted systematic research on failure proximity in Liu et al. (2008) and Liu and Han (2006), in which they summarized or proposed six representative failure proximities, i.e., Failure-based, Stack Trace-based, Code Coverage-based, Predicate Evaluation-based, Dynamic Slicing-based, and Statistical Debugging-based. The CVR utilized in most studies is similar to the above-mentioned Trace-proximity, which has been proven to be less effective in clustering failed test cases. To tackle this limitation, Gao and Wong employed SRR, which is similar to Rank-proximity in Liu et al. (2008), to represent failed test cases. Specifically, (1) they paired each failed test case with all successful test cases and input them into an REF, Crosstab (Wong et al., 2011), to generate a ranking list that represents the corresponding failed test case. (2) They stated that the clustering algorithm's performance highly depends on the distance metric, thus revised the original Kendall tau distance based on the premise that discordant pairs of more suspicious statements contribute more to the distance between two ranking lists. (3) To tackle the long-standing problem of estimating the number of clusters, as well as relieving the uncertainty introduced by randomly generating initial centroids, an approach of selecting initial medoids while predicting the number of clusters was presented inspired by prior studies (Yager and Filev, 1994; Chiu, 1994). (4) They claimed that their initial medoids selection approach reduced the high computational costs to a large extent compared with the original K-medoids clustering algorithm, due to the latter examines all possible combinations of data points as initial medoids. Gao and Wong integrated the above four innovations and developed a novel technique for localizing multiple faults in parallel (Gao and Wong, 2019).

In addition, some researchers have developed a series of novel parallel debugging strategies by integrating techniques from other domains into fault localization. For example, Zakari et al. proposed a fault localization technique that is suited for both single-fault and multi-fault scenarios based on the complex network theory (FLCN), where developers can localize multiple faults at the same time in a single diagnosis ranking list (Zakari et al., 2018). In another study, they adopted the divisive network community algorithm to cluster failed test cases, as well as employed a weighting and selecting mechanism to prioritize generated fault-focused communities (Zakari et al., 2019). Based on one-fault-at-a-time via OPTICS (Ordering Points To Identify the Clustering Structure) clustering, Wu et al. proposed to (1) divide failed test cases in each iteration and calculate the density of each cluster, (2) combine the failed test cases in the cluster with the highest density value with all successful test cases to form a new test suite, and (3) localize a single fault based on the ranking list produced by the new test suite, iterating these steps until all bugs are fixed. Based on their findings, they further concluded that using the clustering algorithm with the highest accuracy can achieve the best performance of multi-fault localization (Wu et al., 2020). Inspired by the multiple-fault-at-a-time strategy, Zheng et al. converted fault localization tasks into search problems and proposed a fast software multi-fault localization framework using genetic algorithms (Zheng et al., 2018). Pei et al. introduced the dynamic random testing (DRT) strategy and proposed distance-based DRT, which vectorized test cases and divided them into disjoint subdomains using distance information from inputs and a specific clustering algorithm (Pei et al., 2021).

There are also some researchers who carried out empirical comparisons of different techniques in the field of multi-fault localization. For instance, Gao et al. contrasted the effectiveness of 22 machine learning algorithms typically used in multi-fault localization and found that random forests, BP neural networks, and logit boost machine learning models based on ensemble learning performed well (Gao et al., 2018). Huang et al. first created 12 types of setup by combining 6 REFs and 2 widely-used clustering algorithms, and then conducted empirical research in multi-fault scenarios using CVR. Their experimental results showed that Wong1 paired with K-means outperformed the other combinations (Huang et al., 2013). Zakari et al. conducted a systematic literature review on classic parallel debugging techniques (Zakari et al., 2020). They investigated off-the-shelf studies and categorized them into three prominent types of strategy, one-fault-at-a-time debugging, parallel debugging, and multiple-fault-at-a-time debugging. Among them, they pointed out parallel debugging alleviated fault interferences through clustering failed test cases. However, many studies such as Jones et al. (2007) and Huang et al. (2013) claimed these existing strategies were insufficient for isolating faults as well as listed some challenges related to clustering effectiveness in parallel debugging, including the method of representing failed test cases, the initial set of fault-focused clusters, the clustering algorithm, and the distance metric.

8. Conclusion and future work

We extract and analyze four essential factors, i.e., the risk evaluation formula that produces ranking lists, the number of faults in a program, the fault types, and the number of successful test cases paired with one individual failed test case, to investigate how these variables affect clustering effectiveness. Four research questions are presented in this paper, the corresponding controlled experiments show that: (1) GP19 is highly competitive across all REFs, thus we recommend that researchers or developers who adopt SRR for parallel debugging use GP19 to represent failed test cases; (2) clustering effectiveness decreases as NOF increases, indicating that a greater number of faults reduces the effectiveness not only in fault localization but also in fault isolation; (3) higher clustering effectiveness is easier to achieve when a program contains only predicate faults, which points out the challenge of isolating assignment faults; and (4) clustering effectiveness remains when NSP1F is reduced to 20%, future researchers and developers are suggested to cut the scale of successful test cases while using SRR for a lower debugging expense.

In the future, we plan to further explore the internal mechanisms of risk evaluation formulas to representing failed test cases, followed by proposing a novel REF for the representation of failed test cases. We also consider investigating the four factors that may influence clustering effectiveness with larger datasets and broader experiment setups, as well as introducing new evaluation metrics.

CRediT authorship contribution statement

Yi Song: Methodology, Software, Draft preparation, Empirical evaluation. **Xiaoyuan Xie:** Methodology, Conceptualization, Supervision. **Quanming Liu:** Software, Data curation. **Xihao Zhang:** Software, Validation. **Xi Wu:** Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the National Key R&D Program of China under the grant number 2020AAA0107803, and the National Natural Science Foundation of China under the grant numbers 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.

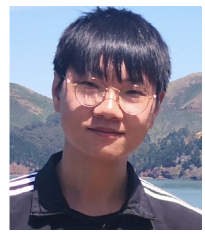
Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2022.111452.

References

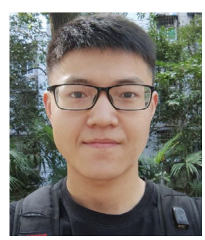
Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2006. An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). IEEE, pp. 39–46.
An, G., Yoon, J., Yoo, S., 2021. Searching for multi-fault programs in defects4j. In: International Symposium on Search Based Software Engineering. Springer, pp. 153–158.
Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering. pp. 402–411.
Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S., 2006. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. 32 (8), 608–624.
Arrieta, A., Segura, S., Markiegi, U., Sagardui, G., Etxeberria, L., 2018. Spectrum-based fault localization in software product lines. Inf. Softw. Technol. 100, 18–31.
Cao, H.L., Jiang, S.J., 2017. Multiple-fault localization based on chameleon clustering. Tien Tzu Hsueh Pao/Acta Electron. Sin. 45 (2), 394–400.
Chen, T.Y., Cheung, S.C., Yiu, S.M., 2020. Metamorphic testing: a new approach for generating next test cases. arXiv preprint arXiv:2002.12543.
Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings International Conference on Dependable Systems and Networks. IEEE, pp. 595–604.
Chiu, S.L., 1994. Fuzzy model identification based on cluster estimation. J. Intell. Fuzzy Systems 2 (3), 267–278.
de Souza, H.A., Chaim, M.L., Kon, F., 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. arXiv preprint arXiv:1607.04347.
DiGiuseppe, N., Jones, J.A., 2011a. Fault interaction and its repercussions. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 3–12.
DiGiuseppe, N., Jones, J.A., 2011b. On the influence of multiple faults on coverage-based fault localization. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 210–220.
DiGiuseppe, N., Jones, J.A., 2012. Concept-based failure clustering. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–4.
DiGiuseppe, N., Jones, J.A., 2015. Fault density, fault types, and spectra-based fault localization. Empir. Softw. Eng. 20 (4), 928–967.
Do, H., Rothermel, G., 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans. Softw. Eng. 32 (9), 733–752.
Fu, W., Yu, H., Fan, G., Ji, X., Pei, X., 2017. A test suite reduction approach to improving the effectiveness of fault localization. In: 2017 International Conference on Software Analysis, Testing and Evolution (SATE). IEEE, pp. 10–19.
Gao, M., Li, P., Chen, C., Jiang, Y., 2018. Research on software multiple fault localization method based on machine learning. In: MATEC Web of Conferences, Vol. 232. EDP Sciences, p. 01060.
Gao, R., Wong, W.E., 2019. MSeer—An advanced technique for locating multiple bugs in parallel. IEEE Trans. Softw. Eng. 45 (3), 301–318.
Golagha, M., Lehnhoff, C., Pretschner, A., Ilmberger, H., 2019. Failure clustering without coverage. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 134–145.
Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L., 2000. An empirical investigation of the relationship between spectra differences and regression faults. Softw. Test. Verif. Reliab. 10 (3), 171–194.
Högerle, W., Steimann, F., Frenkel, M., 2014. More debugging in parallel. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering. IEEE, pp. 133–143.
Huang, Y., Flynt, A., 2018. Exploration of common clustering methods and the behavior of certain agreement indices. Ball State Undergrad. Math. Exch. 12 (1), 35–50.
Huang, Y., Wu, J., Feng, Y., Chen, Z., Zhao, Z., 2013. An empirical study on clustering for isolating bugs in fault localization. In: 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, pp. 138–143.
Jeffrey, D., Gupta, N., Gupta, R., 2008. Fault localization using value replacement. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 167–178.
Jones, J.A., Bowring, J.F., Harrold, M.J., 2007. Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. pp. 16–26.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 273–282.
Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. IEEE, pp. 467–477.
Just, R., Jalali, D., Ernst, M.D., 2014a. 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. pp. 437–440.
Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014b. 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. pp. 654–665.
Kaufman, L., Rousseeuw, P.J., 2009. Finding Groups in Data: An Introduction to Cluster Analysis, Vol. 344. John Wiley & Sons.
Keller, F., Grunske, L., Heiden, S., Filieri, A., van Hoorn, A., Lo, D., 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 (QRS). IEEE, pp. 114–125.
Kendall, M., Gibbons, D., 1990. Rank Correlation Methods. Oxford University Press.
Lamraoui, S.-M., Nakajima, S., 2016. A formula-based approach for automatic fault localization of multi-fault programs. J. Inf. Process. 24 (1), 88–98.
Lee, H.J., Naish, L., Ramamohanarao, K., 2009. Study of the relationship of bug consistency with respect to performance of spectra metrics. In: 2009 2nd IEEE International Conference on Computer Science and Information Technology. IEEE, pp. 501–508.
Lei, Y., Sun, C., Mao, X., Su, Z., 2018. How test suites impact fault localisation starting from the size. IET Softw. 12 (3), 190–205.
Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I., 2005. Scalable statistical bug isolation. Acm Sigplan Notices 40 (6), 15–26.
Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.P., 2006. Statistical debugging: A hypothesis testing-based approach. IEEE Trans. Softw. Eng. 32 (10), 831–848.
Liu, C., Han, J., 2006. Failure proximity: a fault localization-based approach. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 46–56.
Liu, C., Zhang, X., Han, J., 2008. A systematic study of failure proximity. IEEE Trans. Softw. Eng. 34 (6), 826–843.
Mottaghi, N., Keyvanpour, M.R., 2017. Test suite reduction using data mining techniques: A review article. In: 2017 International Symposium on Computer Science and Software Engineering Conference (CSSE). IEEE, pp. 61–66.
Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (3), 1–32.
Pang, Y., Xue, X., Namin, A.S., 2015. Debugging in parallel or sequential: An empirical study. J. Softw. 10 (5), 566–576.
Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2019. Mutation testing advances: an analysis and survey. In: Advances in Computers, Vol. 112. Elsevier, pp. 275–378.
Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, pp. 609–620.
Pei, H., Yin, B., Xie, M., Cai, K.-Y., 2021. Dynamic random testing with test case clustering and distance-based parameter adjustment. Inf. Softw. Technol. 131, 106470.
Perez, A., Abreu, R., van Deursen, A., 2017. A test-suite diagnosability metric for spectrum-based fault localization approaches. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, pp. 654–664.
Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B., 2003. Automated support for classifying software failure reports. In: 25th International Conference on Software Engineering, 2003. Proceedings.. IEEE, pp. 465–475.
Pradel, M., Sen, K., 2018. DeepBugs: A learning approach to name-based bug detection. PACMPL 2 (OOPSLA), 147:1–147:25.
Reps, T., Ball, T., Das, M., Larus, J., 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In: Software Engineering—Esec/Fse'97. Springer, pp. 432–449.

2018. The software infrastructure repository. [Online]. Available: https://sir.csc.ncsu.edu/portal/index.php, accessed on: Nov. 2018. URL https://sir.csc.ncsu.edu/portal/index.php.
Steimann, F., Frenkel, M., 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, pp. 121–130.
Sun, X., Peng, X., Li, B., Li, B., Wen, W., 2016. IPSETFUL: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra. Front. Comput. Sci. 10 (5), 812–831.
Tan, P.-N., Steinbach, M., Kumar, V., 2016. Introduction to Data Mining. Pearson Education India.
Tang, C.M., Chan, W., Yu, Y.T., Zhang, Z., 2017. Accuracy graphs of spectrum-based fault localization formulas. IEEE Trans. Reliab. 66 (2), 403–424.
Wang, Y., Gao, R., Chen, Z., Wong, W.E., Luo, B., 2014. WAS: A weighted attribute-based strategy for cluster test selection. J. Syst. Softw. 98, 44–58.
Wang, X., Jiang, S., Gao, P., Lu, K., Lili, B., Ju, X., Zhang, Y., 2020. Fuzzy C-means clustering based multi-fault localization. Chinese J. Comput. 43 (2), 206–232.
Wang, Q., Wu, S., Li, M., 2008. Software defect prediction. J. Softw. 19 (7), 1565–1580.
Wong, W.E., Debroy, V., Gao, R., Li, Y., 2013. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.
Wong, W.E., Debroy, V., Xu, D., 2011. Towards better fault localization: A crosstab-based statistical approach. IEEE Trans. Syst. Man Cybern. C 42 (3), 378–396.
Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.
Wong, W.E., Qi, Y., Zhao, L., Cai, K.-Y., 2007. Effective fault localization using code coverage. In: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), Vol. 1. IEEE, pp. 449–456.
Wu, Y.-H., Li, Z., Liu, Y., Chen, X., 2020. FATOC: Bug isolation based multi-fault localization by using OPTICS clustering. J. Comput. Sci. Tech. 35 (5), 979–998.
Wu, J., Xiong, H., Chen, J., 2009. Adapting the right measures for k-means clustering. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 877–886.
Xiaobo, Y., Liu, B., Shihai, W., 2018. An analysis on the negative effect of multiple-faults for spectrum-based fault localization. IEEE Access 7, 2327–2347.
Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B., 2013a. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. (TOSEM) 22 (4), 1–40.
Xie, X., Kuo, F.-C., Chen, T.Y., Yoo, S., Harman, M., 2013b. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In: International Symposium on Search Based Software Engineering. Springer, pp. 224–238.
Xie, X., Wong, W.E., Chen, T.Y., Xu, B., 2013c. Metamorphic slice: An application in spectrum-based fault localization. Inf. Softw. Technol. 55 (5), 866–879.
Xie, X., Xu, B., 2021. Essential Spectrum-based Fault Localization. Springer.
Xie, J., Zhou, Y., Wang, M., Jiang, W., 2017. New criteria for evaluating the validity of clustering. CAAI Trans. Intell. Syst. 12 (6), 873–882.
Xu, X., Debroy, V., Eric Wong, W., Guo, D., 2011. Ties within fault localization rankings: Exposing and addressing the problem. Int. J. Softw. Eng. Knowl. Eng. 21 (06), 803–827.
Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M., 2016. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. 43 (1), 34–55.
Xue, X., Namin, A.S., 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, pp. 113–122.
Yager, R.R., Filev, D.P., 1994. Approximate clustering via the mountain method. IEEE Trans. Syst. Man Cybern. 24 (8), 1279–1284.
Yoo, S., 2012. Evolving human competitive spectra-based fault localisation techniques. In: International Symposium on Search Based Software Engineering. Springer, pp. 244–258.
Yoo, S., Xie, X., Kuo, F.-C., Chen, T.Y., Harman, M., 2017. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 26 (1), 1–30.
Yu, Z., Bai, C., Cai, K.-Y., 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, pp. 924–935.
Zakari, A., Lee, S.P., 2019. Parallel debugging: An investigative study. J. Softw.: Evol. Process 31 (11), e2178.
Zakari, A., Lee, S.P., Abreu, R., Ahmed, B.H., Rasheed, R.A., 2020. Multiple fault localization of software programs: A systematic literature review. Inf. Softw. Technol. 124, 106312.
Zakari, A., Lee, S.P., Chong, C.Y., 2018. Simultaneous localization of software faults based on complex network theory. IEEE Access 6, 23990–24002.
Zakari, A., Lee, S.P., Hashem, I.A.T., 2019. A community-based fault isolation approach for effective simultaneous localization of faults. IEEE Access 7, 50012–50030.
Zheng, Y., Wang, Z., Fan, X., Chen, X., Yang, Z., 2018. Localizing multiple software faults based on evolution algorithm. J. Syst. Softw. 139, 107–123.



Yi Song received B.Sc. and M.Phil. degrees in Computer Science from China Three Gorges University, China in 2018 and 2020, respectively. He is currently a Ph.D. candidate in School of Computer Science, Wuhan University, China, under the supervision of professor Xiaoyuan Xie. His research interests mainly include software testing, debugging, program analysis, and search-based software engineering.

Xiaoyuan Xie received B.Sc. and M.Phil. degrees in Computer Science from Southeast University, China in 2005 and 2007, respectively, and received Ph.D. degree in Computer Science from Swinburne University of Technology, Australia in 2012. She is currently a professor in School of Computer Science, Wuhan University, China. Her research interests include software analysis, testing, debugging, and search-based software engineering.



Quanming Liu received B.Sc. degree from Northeastern University at Qinhuangdao, China in 2021. He is currently studying for a master degree in School of Computer Science, Wuhan University, China. His research interests include software testing and fault localization.

Xihao Zhang is currently an undergraduate student of software engineering in School of Computer Science, Wuhan University, China, since 2018. His research interests include software debugging, software fault localization and machine learning.

Xi Wu is currently an undergraduate student of software engineering in School of Computer Science, Wuhan University, China, since 2018. Her research interests include fault localization and artificial intelligence.