Evolving Ranking-Based Failure Proximities for Better Clustering in Fault Isolation

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

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

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

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

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

ABSTRACT

Failures that are not related to a specific fault can reduce the effectiveness of fault localization in multi-fault scenarios. To tackle this challenge, researchers and practitioners typically cluster failures (e.g., failed test cases) into several disjoint groups, with those caused by the same fault grouped together. In such a fault isolation process that requires input in a mathematical form, ranking-based failure proximity (R-proximity) is widely used to model failed test cases. In R-proximity, each failed test case is represented as a suspiciousness ranking list of program statements through a fingerprinting function (i.e., a risk evaluation formula, REF). Although many off-the-shelf REFs have been integrated into R-proximity, they were designed for single-fault localization originally. To the best of our knowledge, no REF has been developed to serve as a fingerprinting function of R-proximity in multi-fault scenarios. For better clustering failures in fault isolation, in this paper, we present a genetic programming-based framework along with a sophisticated fitness function, for evolving REFs with the goal of more properly representing failures in multi-fault scenarios. By using a small set of programs for training, we get a collection of REFs that can obtain good results applicable in a larger and more general scale of scenarios. The best one of them outperforms the state-of-the-art by 50.72% and 47.41% in faults number estimation and clustering effectiveness, respectively. Our framework is highly configurable for further use, and the evolved formulas can be directly applied in future failure representation tasks without any retraining.

*Xiaoyuan Xie is the corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556922

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Search-based software engineering**.

KEYWORDS

Failure proximity, Clustering, Fault isolation, Parallel debugging, Search-based software engineering

ACM Reference Format:
Yi Song, Xiaoyuan Xie, Xihao Zhang, Quanming Liu, and Ruizhi Gao. 2022. Evolving Ranking-Based Failure Proximities for Better Clustering in Fault Isolation. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556922

1 INTRODUCTION

The co-existence of multiple faults in a program can reduce the effectiveness of existing fault localization techniques [12, 70, 77], such as spectrum-based [72] and mutation-based tactics [45]. This is because the majority of these techniques are performed in single-fault scenario, which is considered to be unrealistic due to the increasing scale and complexity of software systems [15, 19, 30, 59]. The crux of multi-fault localization lies in the mutual interference and interaction among faults [65], to put it another way, failed test cases[1] in the test suite (TS) might be linked to distinct root causes. To tackle this challenge, researchers proposed to 1) divide all failed test cases into several disjoint groups according to their root cause [46, 85], with the goal of *having the number of generated groups equal to the number of faults, as well as failed test cases in the same group being triggered by the same fault*, and 2) combine failed test cases in each group with successful test cases for parallel debugging. In the former stage, which is also referred to as **fault isolation** or failure indexing, clustering techniques are typically utilized to achieve the division of failed test cases [13, 22, 52], where unstructured failed test cases need to be converted into a mathematical form before being fed into the algorithm[2]. To that aim, six failure proximities have been summarized or proposed in [41]

[1]The terms "failed test case" and "failure" are interchangeable in this paper.
[2]We quote the terms "clustering" and "division" interchangeably unless otherwise specified hereafter.

for the extraction of formal signatures from failures. Inspired by the ranking-based proximity (R-proximity) among them, statement ranking representation (SRR) is designed and being widely adopted by researchers [7, 20, 62, 82], whose effectiveness and promise have been demonstrated by extensive trials [20, 39].

While utilizing SRR to preprocess failures, a failed test case is represented as a ranking list of statements built upon their likelihood of being faulty. Specifically, a failure-specific TS that comprises a failed test case and successful test cases is executed on the program under test (PUT), along with the coverage information being collected [23, 49]. The resulting spectrum data is then sent to a fault localization technique, which is generally a risk evaluation formula (REF) in spectrum-based fault localization (SBFL) [72, 80], to produce a failure-specific ranking list that represents this failed test case in a clustering-friendly form.

A large number of existing REFs, including both manually customized ones (such as Tarantula [29], Ochiai [1], and Crossrab [64]) and genetic programming evolved ones (such as GP02, GP03, and GP19 [79]), were all designed to localize single-fault, i.e., based on to what extent they can push the faulty statement to the top of the ranking list, rather than serve as a fingerprinting function of R-proximity in multi-fault scenarios. But according to our investigation, almost all researchers who have employed R-proximity used a specific off-the-shelf REF to represent failures directly [20, 28, 39].

However, **the factors that need to be considered in multiple-fault isolation and single-fault localization are not exactly the same.** Therefore, the fault localization optimal REF in the single-fault scenario does not necessarily imply good performance in multi-fault isolation. It is intuitive that if an REF has a stronger capability to extracting signatures from failures, the ranking lists it produces will more properly model failed test cases, thus better parallel debugging effectiveness can be obtained. Therefore, delivering an REF that is capable of properly representing failures in multi-fault scenarios is non-trivial. To the best of our knowledge, no research has been done to design an REF for such a motivation.

In this paper, we are interested in investigating whether there is any REF that can deliver better performance than currently adopted ones, in terms of fault isolation. We present a genetic programming-based framework, together with a sophisticated fitness function, to automatically evolve formulas with the goal of more effectively extracting signatures from failures in multi-fault scenarios.

We download four projects written in C (flex, grep, gzip, and sed) from the Software Infrastructure Repository (SIR) [50], followed by creating 960 faulty versions through artificially injecting faults into clean programs. These injected faults are of varied numbers (i.e., two, three, four, and five) and varied types (i.e., assignment fault and predicate fault). A collection of risk evaluation formulas are evolved on a small set (i.e., 15%) of these C programs and then **tested on both the remaining C programs as well as real-world Java programs**. Experiments reveal that in the context of failure representation, evolved fingerprinting functions (EFF)[3] are highly human-competitive: a substantial number of EFFs can not only outperform all existing REFs on simulated C faults, but also obtain similar results to existing REFs on real-world Java faults. Among them, EFF10-83 exceeds the state-of-the-art REF the most, with increases of 50.72% and 47.41% regarding faults number estimation and clustering effectiveness, respectively[4].

This paper makes the following contributions:

- Instead of designing a risk evaluation formula with higher single-fault localization effectiveness, in this paper, we construct fingerprinting functions of R-proximity for better representing failures in multi-fault isolation. As far as we are aware, this is the first work to develop risk evaluation formulas from this perspective.
- Rather than manually design the formula, we adopt genetic programming to automatically evolve formulas without any interference from human beings. The proposed configurable evolution strategy, along with the sophisticated fitness function, provides a reasonable way to generate and evaluate R-proximity fingerprinting functions, which enables future researchers to evolve better ones on their own.
- Using the proposed approach, we successfully evolve formulas that are competitive with previously human-designed ones. Based on the results, we recommend that stakeholders who adopt R-proximity employ EFF10-83, the most effective individuals in the experiment, as the fingerprinting function.

The remainder of this paper is organized as follows: Section 2 reports background and gives a motivating example. Section 3 describes our evolution framework and the fitness function. Section 4 provides the parameter setting, datasets, metrics, and experimental environments. Section 5 analyzes the experimental results. Section 6 further discusses our approach. Section 7 declares the threats to validity. Section 8 summarizes the related work. Conclusions and directions for future work are proposed in Section 9.

2 BACKGROUND

2.1 Failure Representation

Failure representation is an essential step in fault isolation[5]. This is because failed test cases are typically too abstract to be directly used in the clustering process, it is necessary to translate them into a mathematical and structured form. To that end, Liu et al. conducted systematic research on failure proximity in [39] and [41], in which they summarized or proposed six representative failure proximities, namely, failure-based, stack trace-based, code coverage-based, predicate evaluation-based, dynamic slicing-based, and statistical debugging-based ones. Coverage vector representation (CVR), which is similar to the trace-proximity, creates a vector with a length equal to the number of executable statements in the PUT, for representing a failed test case. In such a vector, the value of the i^{th} element is set to 1 if this failed test case covers the i^{th} statement, and 0 otherwise. Although CVR has been utilized in a significant number of previous research such as [14, 24, 83], it has proven to be problematic since a fault can be triggered in a variety of ways [20, 83]. SRR, which is similar to the rank-proximity, is in widespread use for its promise of capturing execution signatures of failures via fault

[3] An evolved fingerprinting function (EFF) is also a risk evaluation formula (REF). In this paper, in order to facilitate the comparison and analysis, we denote the off-the-shelf risk evaluation formula designed for single-fault localization as REF, while denote the evolved REF that serves as a fingerprinting function of R-proximity as EFF.

[4] We share the source code and data at: https://github.com/yisongy/SRR-GP.

[5] Because multiple faults are typically isolated by indexing failures to their root cause, "fault isolation" is also referred to as "failure indexing".

Table 1: 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^{**}	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 x, y, z	·	·	·	·	·	·	·	·	5/3/2	0/0/0	3/3/3	0/0/0	8/6/5	0/0/0	3/3/3	5/3/2	8/6/5	**0.79**	0.71	0.63
s_2	if (x < y):	·	·	·	·	·	·	·	·	5/3/2	0/0/0	3/3/3	0/0/0	8/6/5	0/0/0	3/3/3	5/3/2	8/6/5	**0.79**	0.71	0.63
s_3	if (x < z):	·			·		·	·		3/3/0	2/0/2	0/1/1	3/2/2	3/4/1	5/2/4	3/3/3	5/3/2	8/6/5	0.77	0.87	0
s_4	s = x + y //$Fault_1$ ✓s = y + z	★			★			★		3/3/0	2/0/2	0/0/0	3/3/3	3/3/0	5/3/5	3/3/3	5/3/2	8/6/5	0.77	**1**	/
s_5	else:						·			0/0/0	5/3/2	1/1/1	2/2/2	1/1/1	7/5/4	3/3/3	5/3/2	8/6/5	0	0	0
s_6	z = x + y						·			0/0/0	5/3/2	1/1/1	2/2/2	1/1/1	7/5/4	3/3/3	5/3/2	8/6/5	0	0	0
s_7	else:		·	·		·			·	2/0/2	3/3/0	2/2/2	1/1/1	4/2/4	4/4/1	3/3/3	5/3/2	8/6/5	0.45	0	0.71
s_8	if (z <y)		·	·		·			·	2/0/2	3/3/0	2/2/2	1/1/1	4/2/4	4/4/1	3/3/3	5/3/2	8/6/5	0.45	0	0.71
s_9	s = x * z //$Fault_2$ ✓s = x + y			○					○	2/0/2	3/3/0	0/0/0	3/3/3	2/0/2	6/6/3	3/3/3	5/3/2	8/6/5	0.63	/	**1**
s_{10}	else		·			·				0/0/0	5/3/2	2/2/2	1/1/1	2/2/2	6/4/3	3/3/3	5/3/2	8/6/5	0	0	0
s_{11}	s = x + z		·			·				0/0/0	5/3/2	2/2/2	1/1/1	2/2/2	6/4/3	3/3/3	5/3/2	8/6/5	0	0	0

localization techniques (i.e., a risk evaluation formula in SBFL), instead of being simply based on covering paths [20, 36, 41].

2.2 SBFL Notations

Coverage information is typically referred to as binary vectors, which indicate whether a program statement[6] is covered by a test case during running a TS on a PUT. To formalize these binary indicators, researchers in the field of SBFL defined nine notations that are capable of revealing execution characteristics [56], that is, N_{CF}, N_{CS}, N_{UF}, N_{US}, N_C, N_U, N_S, N_F, and N (also referred to as a_{ef}, a_{ep}, a_{nf}, a_{np}, a_e, a_n, a_p, a_f, and a, respectively), where N_{CF} and N_{CS} represent the number of test cases that execute the statement and return the testing result of failed or successful, respectively, N_{UF} and N_{US} represent the number of test cases that do not execute it and return the testing result of failed or successful, respectively, N_C and N_U represent the number of test cases that execute and do not execute the statement, respectively, N_S and N_F represent the number of successful and failed test cases, respectively, N is the scale of the test suite. The intuition of designing SBFL formulas for single fault localization is that statements associated with more failed and less successful testing results are more likely to be faulty [43, 54, 74].

2.3 Motivating Example

We exemplify fault isolation and the SRR strategy in parallel debugging through a motivating example.

As shown in Table 1, the PUT that contains 11 statements is designed to calculate the sum of the bigger two of the three numbers, in which two faults have been induced by statements s_4 and s_9, respectively. Given a TS with eight test cases: t_1 = {3,8,6}, t_2 = {7,5,8}, t_3 = {9,6,2}, t_4 = {5,7,6}, t_5 = {2,1,5}, t_6 = {6,9,4}, t_7 = {4,8,7}, and t_8 = {9,6,1}, five of them are labelled as failed owing to unexpected outputs (t_1, t_3, t_4, t_7, t_8). The 11×8 matrix composed of rows s_1 to s_{11} and columns t_1 to t_8 in Table 1 is the coverage information. This matrix is obtained by running the TS against the given PUT, where $t_1 \sim t_8$ columns reflect the execution paths of the eight test cases. The symbol "·" implies that a test case covers an innocent statement, whereas "★" and "○" imply that a test case covers the statements containing $Fault_1$ or $Fault_2$, respectively. The coverage information is reorganized into spectrum information according to the notations defined in SBFL, as shown in the 11×9 matrix composed of rows s_1 to s_{11} and columns N_{CF} to N in Table 1[7].

[6]Unless otherwise specified, "statement" refers to "executable statement" in this paper.
[7]Each cell in the column "Spectrum Information" is in the form of "a/b/c", where a, b, and c represent the value of notation under $F_t \cup S$, $F_1 \cup S$, and $F_2 \cup S$, respectively. And in the column "Suspiciousness", the suspiciousness of s_9 under $F_1 \cup S$ and the suspiciousness of s_4 under $F_2 \cup S$ cannot be calculated, since Ochiai loses its definition when the value of N_C is zero.

2.3.1 Fault Isolation in Parallel Debugging. In this example, it can be observed that five failed test cases are triggered by two distinct root causes, $Fault_1$ and $Fault_2$. If they are not divided properly, fault localization techniques could be confused by the comprehensive test suite significantly. For example, SBFL techniques will extract execution features of both two faults guided by the messy spectrum data, which will lower the rank of each fault in the generated ranking list. Let us employ a well-known REF, Ochiai [1], as defined in Formula 1, to illustrate how such a detrimental effect harms the fault localization process.

$$suspiciousness_{\text{ Ochiai}} = \frac{N_{CF}}{\sqrt{N_F N_C}} \tag{1}$$

We first calculate each statement's suspiciousness by following the traditional strategy, that is, leveraging all failed test cases F_t and all successful test cases S to activate Ochiai. The results are shown in column $F_t \cup S$ in Table 1. We can immediately sort program statements in descending order of suspiciousness and get a ranking list: {s_1, s_2, s_3, $\boldsymbol{s_4}$, $\boldsymbol{s_9}$, s_7, s_8, s_5, s_6, s_{10}, s_{11}}. In this list, the statement s_4 containing $Fault_1$ and the statement s_9 containing $Fault_2$ are ranked fourth and fifth, respectively. Three innocent statements, s_1, s_2, and s_3, will be inspected before s_4 and s_9.

In five failed test cases in TS, t_1, t_4, and t_7 (with superscript "*" in Table 1) are triggered by $Fault_1$, while t_3 and t_8 (with superscript "**" in Table 1) are triggered by $Fault_2$. In fault isolation, ideally, they should be rearranged into two fault-focused clusters, namely, F_1 = {t_1, t_4, t_7}, and F_2 = {t_3, t_8}. 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, respectively. After executing these two fault-focused TSs on the PUT, two sets of spectrum data can be collected, as shown in Table 1. The suspiciousness of statements calculated by Ochiai using these two sets of spectrum data is shown in columns $F_1 \cup S$ and $F_2 \cup S$ in Table 1, respectively. Two faulty statements, s_4 and s_9, are both found to be riskiest in the corresponding ranking list, enabling two independent developers to effectively debug the two faults in a parallel manner.

We can conclude two essential points from this toy example: 1) fault isolation can alleviate the negative impact caused by the multi-fault co-existence, thus making the fault localization technique deliver more pertinent outcomes, and 2) the core of fault isolation lies in the division of failed test cases. As stated previously, clustering techniques are typically employed to serve as a division

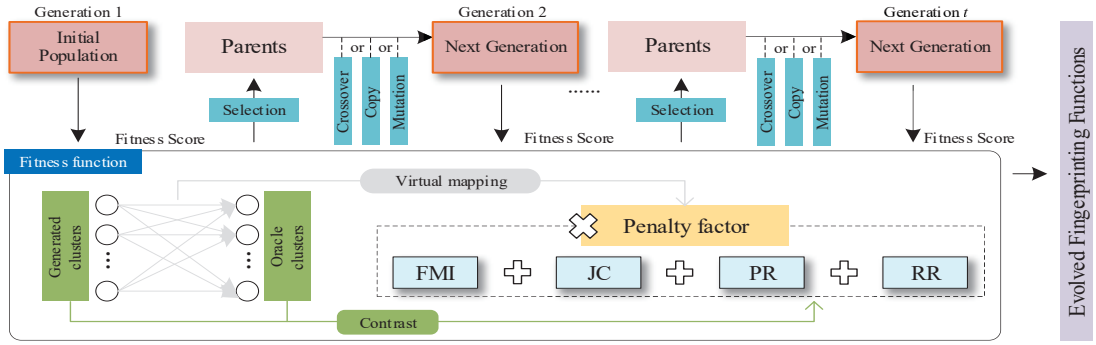

Figure 1: The overview of the proposed framework

strategy, where unstructured failed test cases need to be converted to a mathematical form, i.e., failure representation. The SRR strategy, one of the most sophisticated and advanced failure proximities for this purpose, is illustrated as follows.

2.3.2 Statement Ranking Representation. Take t_3 as an example. First, pairing t_3 with S to form a failure-specific TS, $t_3 \cup S$. Second, executing this TS on PUT to obtain coverage information, and then converting the coverage into spectrum data. Third, utilizing a risk evaluation formula (e.g., Ochiai) to determine each statement's suspiciousness. Finally, a ranking list can be produced to represent t_3, i.e., {4, 4, 6, 11, 6, 6, 2, 2, 1, 6, 6}[8], which will be invoked in the subsequent clustering process as a proxy for t_3. Similarly, t_7 can be represented as a ranking list {3, 3, 2, 1, 5, 5, 5, 5, 11, 5, 5}. To elucidate the distinction in capabilities to representing failed test cases across different REFs, we produce proxies for t_3 and t_7 using another REF, Naish2, as defined in Formula 2. The outputs are {4, 4, 7, 6, 7, 7, 2, 2, 1, 10, 10} and {3, 3, 2, 1, 6, 6, 8, 8, 5, 8, 8}, respectively.

$$suspiciousness_{\text{Naish2}} = N_{CF} - \frac{N_{CS}}{N_S + 1} \tag{2}$$

The value of distance (e.g., Euclidean distance) between t_3 and t_7 (using Ochiai) is calculated as being 15.49 while being 12.25 (using Naish2)[9]. Obviously, Ochiai distinguishes the two failed test cases more properly than Naish2. However, the latter has theoretically proven to be better than the former in terms of single-fault localization [72]. As a consequence of this observation, we can conjecture that a stronger REF in single-fault localization is not necessarily dominant when it comes to serving as a fingerprinting function of R-proximity for failure representation in multi-fault scenarios.

Notice that in this motivating example: 1) the execution paths of the failed test cases triggered by the same fault are also the same, and 2) a failed test case has only one root cause. Such a simplification (or an assumption) is only used to make the example more understandable while is not prevalent in practice. In real-world fault isolation tasks: 1) a fault could be triggered in different ways, and 2) more than one fault could be responsible for a failure at the same time. Obviously, multiple faults could be more difficult to be isolated in this scenario than in the previous simple assumption. In this paper, for a more practical and robust evaluation, we adopt the more realistic assumption rather than the simplified one.

[8]In this phase, if several statements with the same suspiciousness form a tie [75], the rankings of all statements in the tie will be set to the beginning position of this tie.
[9]The subsequent clustering process is omitted due to the limited space.

3 APPROACH

In this section, we outline the overview of the proposed evolution framework in Section 3.1, introduce the fitness function in Section 3.2, and describe key steps of the evolution process in Section 3.3.

3.1 Overview

We employ genetic programming (GP) [61] as the evolving algorithm owing to two points. First, a risk evaluation formula is made up of notations (i.e., N_{CS}, N_{CF}, N_{US}, and N_{UF}) and various operators, the possible combinations between them are numerous and complicated, resulting in a big search space, and GP is well-suited to handle such a difficulty [2]. Second, in SRR-based fault isolation, the obtaining of clustering results involves several phases, thus the relationship between an EFF and its failure representation capability cannot be simply determined, GP can relieve this challenge through a well-designed fitness function.

The workflow of the proposed framework is illustrated in Figure 1. The details of the workflow are formally outlined in Algorithm 1, where *IP*, *PA*, *NG*, and *FS* represent the *Initial Population*, *Parents*, *Next Generation*, and *Fitness Score* in Figure 1, respectively, and *CP* represents the current population. The inputs of the framework include six hyperparameters (n, d, p, c, o, and m), one external method (*FF*), and a stopping criterion (*SC*), with the evolved fingerprinting functions as the output.

3.2 Fitness Function

The calculation of the fitness score of an EFF equals the evaluation of the clustering process based on it, which can be measured by two factors, *Total_metrics* and *PenaltyFactor*. *Total_metrics* involves four external metrics that are commonly used in the evaluation of a clustering process [67, 71], i.e., the Fowlkes and Mallows Index (FMI), the Jaccard Coefficient (JC), the Precision Rate (PR), and the Recall Rate (RR), which are defined and introduced in Section 3.2.1. *PenaltyFactor* is designed to address the threats posed by the virtual mapping problem and to heuristically guide the evolution, which is defined and introduced in Section 3.2.2. The fitness score of an EFF on **one** faulty version is visualized in the lower part of Figure 1 and defined in Formula 3.

Algorithm 1 Evolution Framework

Input:
the size of population $\boldsymbol{n}$, the maximum tree depth of EFF $\boldsymbol{d}$, fitness function $\boldsymbol{FF}$, the size of selected parents $\boldsymbol{p}$, crossover rate $\boldsymbol{c}$, copy rate $\boldsymbol{o}$, mutation rate $\boldsymbol{m}$, stopping criterion $\boldsymbol{SC}$
Output:
evolved fingerprinting function
1: $\boldsymbol{IP} \leftarrow \boldsymbol{n}$ randomly initialized EFFs within the constraint of $\boldsymbol{d}$
2: $\boldsymbol{CP} \leftarrow \boldsymbol{IP}$
3: **repeat**
4: $\boldsymbol{FS} \leftarrow \boldsymbol{FF}(\boldsymbol{CP})$
5: $\boldsymbol{PA} \leftarrow$ Selecting $\boldsymbol{p}$ individuals in $\boldsymbol{CP}$ according to $\boldsymbol{FS}$
6: $\boldsymbol{NG} \leftarrow$ crossover, copy, or mutate individuals in $\boldsymbol{PA}$ with $\boldsymbol{c}$, $\boldsymbol{o}$, and $\boldsymbol{m}$, respectively
7: $\boldsymbol{CP} \leftarrow \boldsymbol{NG}$
8: **until** $\boldsymbol{SC}$ == True

$$FitnessScore = Total\_metrics * PenaltyFactor \tag{3}$$

Given an EFF, and a program that contains r faults (referred to as an r-bug faulty version) and corresponding test cases, the fitness function is determined by four factors (DM, EN, IM, and CA), and the output is fitness score (FS), as formally depicted in Algorithm 2.

Algorithm 2 Fitness Function

Input:
an r-bug faulty version, an evolved fingerprinting function $\boldsymbol{EFF}$, distance metric $\boldsymbol{DM}$, the faults number estimation strategy $\boldsymbol{EN}$, the medoids initialization strategy $\boldsymbol{IM}$, clustering algorithm $\boldsymbol{CA}$
Output:
fitness score
1: Utilize $\boldsymbol{EFF}$ to convert failed test cases into ranking lists
2: Calculate distances between ranking lists using $\boldsymbol{DM}$
3: Employ $\boldsymbol{EN}$ to get the predicted number of faults k
4: **If** k != r:
5: set $\boldsymbol{FS}$ to be zero
6: **Elif** k == r:
7: Leverage $\boldsymbol{IM}$ to initialize medoids and running $\boldsymbol{CA}$
8: Obtain $Total\_metrics$ and $PenaltyFactor$ by calculating and analyzing four external metrics
9: set $\boldsymbol{FS}$ to be $Total\_metrics$ times $PenaltyFactor$

Notice that in the designed fitness function, we validate the EFF's fitness score only when the faulty version meets the requirement of "$k == r$", i.e., the estimated number of clusters is equal to the number of faults, otherwise the fitness score is set to zero. The reason behind this scheme lies in the goal of fault isolation, that is, all failed test cases need to be divided into several groups with each of which targeting a single fault. Only when the number of generated clusters is equal to the number of faults can we properly establish mapping relations between them. Actually, in practical parallel debugging, "$k \neq r$" is a more common scenario since properly representing and dividing failures is indeed tricky. The subsequent localization step can also run when k is not equal to r, but this may incur unnecessary costs. We further discuss this in Section 6 and Section 7.

Table 2: Scenarios in two types of metrics

Metric	Notation	Results of failure indexing	
		In generated cluster	In oracle cluster
pair of cases-based	SS	Same	Same
	SD	Same	Difference
	DS	Difference	Same
	DD	Difference	Difference
single case-based	TP	Positive	Positive
	FP	Positive	Negative
	TN	Negative	Negative
	FN	Negative	Positive

3.2.1 Total_metrics. External metrics [67] and internal metrics [53] are generally implemented to measure a clustering process, with the former being preferred when the oracle is accessible. In fault isolation, the oracle is described as the real linkages between failed test cases and faults, which can be compared with the generated clusters. Considering the availability of such oracles, we select four widely-used external metrics, FMI, JC, PR, and RR, to quantitatively evaluate the clustering in our fitness function. Among them, we refer to FMI and JC as *pair of cases-based metrics*, and refer to PR and RR as *single case-based metrics*.

Pair of cases-based metrics compare the indexing consistency of each pair of failed test cases in the generated cluster with that in the oracle cluster. Four possible scenarios in this process are depicted in Table 2. Suppose there are n failed test cases, a total of C_n^2 pairs will be examined. The numbers of pairs that fall into SS, SD, DS, and SS categories are denoted by X_{SS}, X_{SD}, X_{DS}, and X_{DD}, respectively, and these four notations can be incorporated into FMI and JC, which are defined in Formula 4 and Formula 5, respectively. FMI and JC are used to determine the similarity between the generated cluster and the oracle cluster [25], the larger the value in their interval, i.e., [0, 1], the more effective clustering is.

$$FMI = \sqrt{\frac{X_{SS}}{X_{SS}+X_{SD}} \times \frac{X_{SS}}{X_{SS}+X_{DS}}} \tag{4}$$

$$JC = \frac{X_{SS}}{X_{SS}+X_{SD}+X_{DS}} \tag{5}$$

Single case-based metrics compare the classification result of each failed test case in the generated cluster with that in the oracle cluster. Four possible scenarios in this process are depicted in Table 2. We use X_{TP}, X_{FP}, X_{TN}, and X_{FN} to denote the numbers of failed test cases that fall into TP, FP, TN, and FN categories, respectively, and these four notations can be incorporated into PR and RR, which are defined in Formula 6 and Formula 7, respectively. The intervals of PR and RR are both [0, 1] and that the larger the value in this range, the more effective clustering is.

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

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

Different permutations of generated and oracle clusters will result in different external metric outputs, and the diversity of permutations will expand dramatically as the number of faults grows. For example, in a 2-bug faulty version, the permutations of two generated clusters and two oracle clusters are $A_2^2 = 2$, while in a 5-bug faulty version, the permutations of 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 exists in the contrast between the clustering output and the oracle in experiments. In other words, each developer will be allocated to a fault-focused TS, and they will be responsible for localizing the corresponding fault independently. Regardless of how many underlying *permutations* exist, there is **only one** real *combination* of generated clusters and oracle clusters (we call this problem the virtual mapping problem). For those "$k == r$" faulty versions, we first enumerate all feasible permutations, and then pick the optimal one depending on the value of FMI, JC, PR, or RR for evaluation, because which permutation reflects the real mapping relations is unknown.

By doing so, we get the optimal value of each metric, denoted as FMI_{opt}, JC_{opt}, PR_{opt}, and RR_{opt}. We design $Total\_metrics$ to incorporate them, as shown in Formula 8. The value of $Total\_metrics$ of an EFF on a faulty version is in the range of [0, 4].

$$Total\_metrics = FMI_{opt} + JC_{opt} + PR_{opt} + RR_{opt} \tag{8}$$

3.2.2 PenaltyFactor. While evaluating the effectiveness of an EFF on a faulty version, the optimal value of four metrics might occur on different permutations, we believe that the consistency among them reflects the **rationality** of the contrast between generated and oracle clusters. Consequently, we also take such a consistency into consideration since it associates with $Total\_metrics$. Specifically, if the optimal values of the four metrics all appear on the same permutation, it means that the four metrics can easily reach a consensus, indicating that the ranking lists produced by the EFF represent failed test cases distinguishably. On the contrary, if the optimal values of the four metrics are dispersed onto different permutations, such a divergence indicates that the ranking lists produced by the EFF are too analogous to be divided.

We regard the evaluation of the four metrics for all permutations as a voting process, in which each metric votes for the permutation with its highest value. Thus, a permutation will get four votes if the highest values of all four metrics appear on it. In an r-bug version, A_k^r permutations will each be assigned a value of vote, and this r-bug version's vote will be determined as the highest value of vote among A_k^r permutations (denoted as $Vote_{most}$). We design $PenaltyFactor$ as an extra weight, to lower the fitness score of a fault isolation process with **less rationality**, as shown in Formula 9.

$$PenaltyFactor = 1 - 0.05 * (4 - Vote_{most}) \tag{9}$$

The possible values of $PenaltyFactor$ are 0.85, 0.9, 0.95, and 1, when $Vote_{most}$ takes 1, 2, 3, and 4, respectively.

3.3 Evolution Processes

The evolution process of our framework is threefold, which involves initial population generation, selection, as well as crossover, copy, and mutation, as shown in the upper part of Figure 1.

3.3.1 Initial Population Generation. The initial population generation is performed by repeatedly creating a single EFF until the size of the population reaches the predefined threshold. Our framework constructs an EFF by integrating spectrum notations (i.e., N_{CF}, N_{CS}, N_{UF}, and N_{US})[10] and fundamental integers (i.e., 1 and -1) with several operators (e.g., addition, subtraction, multiplication, division, negation, exponential operation, and tanking the absolute value). In a tree-based GP model, the generation of an EFF is accomplished by constructing a tree, with the leaf nodes being spectrum notations or fundamental integers and the branch nodes being operators. For instance, the EFF shown in Formula 10 is generated by constructing a tree shown in Figure 2. The crossover and mutation of EFFs are both performed by altering their corresponding trees.

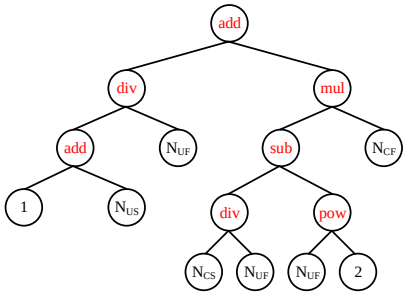

Figure 2: EFF in tree form

$$\frac{N_{US}+1}{N_{UF}} + \left(\frac{N_{CS}}{N_{UF}} - N_{UF}^2\right) \times N_{CF} \tag{10}$$

3.3.2 Selection. During the evolution, we employ the roulette algorithm to select a random subset of the current population according to their fitness scores, for serving as the parents for the next generation. The intuition behind such a strategy is that the higher the fitness score of an EFF, the more suitable it is 1) for serving as a fingerprinting function of R-proximity, and 2) for evolving better individuals through crossover, copy, or mutation. The scale of the parents should be set properly since if it is too large, many EFFs with lower fitness scores will be selected, resulting in a decline in overall population fitness, thus hindering an efficient evolution. On the contrary, if the scale of the parents is too small, there will be many redundant individuals in the next generation, which can lead to a decrease in overall population richness.

In particular, to prevent the evolution framework from searching in a recursive space, if an EFF in the n^{th} generation is consecutively selected and copied into the $n+1^{\text{th}}$ and $n+2^{\text{th}}$ generations, it is no longer permitted to appear in the $n+3^{\text{th}}$ generation.

3.3.3 Crossover, Copy and Mutation. To develop EFFs for the next generation, our framework first determines the evolving strategy (i.e., crossover, copy, or mutation) according to the preset crossover rate, copy rate, and mutation rate, respectively, and then generates a new EFF by constructing a tree based on the existing EFF/EFFs (EFF for copy and mutation, and EFFs for crossover) in the parents generation. This process will be continued until the population scale reaches the predetermined level.

To guarantee overall population richness, we stipulate that the same EFF/EFFs in the parents generation cannot be used to generate individuals for the next generation through the same evolving

[10]The other spectrum notations N_C, N_U, N_S, N_F, and N could be obtained by the simple integration of these four basic elements.

strategy. For example, if an EFF has already been generated by mutating EFF P_1 in the parents generation, our framework prevents further mutations to EFF P_1, but permits building a new EFF by 1) applying crossover or copy to EFF P_1, or 2) mutating the other EFFs in the parents generation other than EFF P_1.

4 EXPERIMENTAL SETUP

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

4.1 Parameter Setting

As mentioned in Section 3.1, the inputs of the evolution framework involve six hyperparameters, one external method (fitness function), and a stopping criterion. In our experiments, we set 160 as the size of population $\boldsymbol{n}$, 17 as the maximum tree depth $\boldsymbol{d}$, 80 as the size of selected parents $\boldsymbol{p}$, and 0.7, 0.2, and 0.1 as the (single-point) crossover rate $\boldsymbol{c}$, copy rate $\boldsymbol{o}$, and mutation rate $\boldsymbol{m}$, respectively. The fitness function $\boldsymbol{FF}$ we designed is illustrated in Section 3.2. The stopping criterion $\boldsymbol{SC}$ is configured to a fixed run of 15 generations, considering earlier studies' experience [21, 51, 79].

As mentioned in Section 3.2, the inputs of the fitness function involve four factors. We adopt the euclidean distance as the distance metric $\boldsymbol{DM}$, the solutions of MSeer [20] that are based on the mountain method [9, 78] as the faults number estimation strategy $\boldsymbol{EN}$ and the medoids initialization strategy $\boldsymbol{IM}$, and k-medoids [33] as the clustering algorithm $\boldsymbol{CA}$.

The above hyperparameters are not hard-coded but can be configurable, determining a specific value or choice for each of them (no matter optimal or not) just enables the framework to run, thus we can evaluate whether our framework is promising and competent to its mission.

4.2 Datasets

We create 960 C programs each with multiple artificial faults for the training and test. We also create 100 Java programs each with multiple real-world faults only for the test, to examine the practicability and applicability of the evolved formulas.

4.2.1 C Programs (SIR). We download four classic projects from SIR [50]: flex, grep, gzip, and sed, as shown in Table 3. Research such as [4, 5, 16, 32, 38, 48] has confirmed that mutation-based faults can simulate real-world faults to an extent, thus can provide credible results for experiments in the field of software testing and debugging. In light of that, we use these four projects as benchmark programs to generate 228 single-bug faulty versions by employing mutation-based strategies [44]. Specifically, we adopt an existing tool [6] with 10 "fork" and 21 "star" on GitHub to perform mutation. It defines 67 types of point that can be mutated, and provides several mutation operators for each one. The mutation operators we leverage (i.e., the fault types we investigate) can be categorized into the following two classes.

- ***Assignment Fault*** [27]: Editing a variable's value in the statement, or replacing the operators such as addition, subtraction, multiplication, division, etc. with each other.
- ***Predicate Fault*** [76]: Reversing the $if\text{-}else$ predicate, or deleting the *else* statement, or modifying the decision condition, and so on.

Table 3: Subject Programs

Project	Version	kLOC	No. of faults	Description
flex	2.5.3	14.5	76	Parser generator
grep	2.4	13.5	47	Text matcher
gzip	1.2.2	7.3	44	File archiver
sed	3.02	10.2	61	Stream editor
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
Math	2.0.0	85.5	29	Apache commons-math
Time	2.0.0	28.4	20	Date and time library

To create an r-bug faulty version, the faults from r individual single-bug faulty versions are injected into the same program, such a technique has been implemented in lots of previous research [26, 34, 82]. If a multi-fault program contains only *assignment* faults or only *predicate* faults, we refer to it as a TypeA faulty version or a TypeP faulty version, respectively. A TypeH faulty version is a program in which two types of fault occur *hybridly*. A total of 960 multi-fault versions are generated in our experiments, which can be categorized dually: it consists of 2-bug, 3-bug, 4-bug, and 5-bug ones evenly (i.e., 240 of each) from the perspective of the number of faults, while consists of TypeA, TypeP, and TypeH ones evenly (i.e., 320 of each) from the perspective of fault types. Among them, 15 percent are expropriated for the training and the remaining 85 percent are used for the test.

4.2.2 Java Programs (Defects4J). Defects4J is one of the most popular benchmarks in the current field of fault localization [31], but it is generally used in single-fault scenarios, because each of its faulty versions only targets a specific fault. Recently, An et al. adapted Defects4J to multi-fault scenarios by transplanting the fault-revealing test case(s) of other faulty version(s) to a basic faulty version, enabling a strengthened test suite to detect more faults in the original program (i.e., the basic faulty version) [3]. Following their strategy, a total of 100 multi-fault versions are generated using 141 faults on five projects, i.e., Chart, Closure, Lang, Math, and Time, as shown in Table 3. The generation of Defects4J multi-fault versions is based on the search in real-world software development, not manual. Because of this, the number of faulty versions that can be generated is limited, and specifying the number of faults and the fault type for each faulty version could be difficult. It is challenging to complete the training phase on such small-scale and unpolished datasets. Therefore, all Defects4J benchmarks are only used for the test.

Because the way of combining multiple single-faults in Defects4J differs from that in SIR, and faults provided in Defects4J are from the real world, the numbers of generated 2-bug, 3-bug, 4-bug, and 5-bug Defects4J faulty versions are unbalanced, and we can hardly categorize them as TypeA, TypeP, or TypeH. Thus, we present the test results on Defects4J programs in a separate part in Section 5.

4.3 Metrics

The strategy for calculating an EFF's fitness score on one faulty version has been given in Algorithm 2 and Formula 3. In our experiments, we determine an EFF's fitness score by simply adding

Table 4: The clustering effectiveness of 12 groups of existing risk evaluation formulas on SIR

		Group1	Group2	Group3	Group4	Group5	Group6	Group7	Group8	Group9	Group10	Group11	Group12
2-bug	FV_e	9	**13**	5	**49**	11	8	10	11	11	12	10	**25**
	Sum_{FS}	22.65	**35.91**	15.84	**150.16**	31.43	22.79	24.9	26.91	31.46	35.41	34.16	**74.48**
3-bug	FV_e	30	**35**	16	29	**40**	28	31	22	32	20	24	**34**
	Sum_{FS}	82.18	**93.63**	43.14	79.51	**107.69**	76.84	85.22	59.36	86.54	54.24	66.39	**90.91**
4-bug	FV_e	24	16	**33**	**34**	16	15	21	30	17	19	**44**	24
	Sum_{FS}	59.4	41.52	**82.07**	**82.21**	40.07	37.98	51.03	74.02	43.14	47.53	**105.59**	59.33
5-bug	FV_e	19	22	**26**	**26**	19	11	21	15	11	**33**	**29**	**26**
	Sum_{FS}	46.11	53.21	62.05	60.36	46.3	27.27	51.08	35.52	27.54	**77.56**	**67.98**	**62.45**
TypeA	FV_e	23	27	22	**36**	26	23	23	24	25	22	**39**	**35**
	Sum_{FS}	59.66	72.76	56.72	**95.51**	69.91	62.54	59.35	58.45	67.72	57.58	**98.41**	**96**
TypeP	FV_e	27	26	28	**46**	30	17	27	23	21	34	**39**	**37**
	Sum_{FS}	70.68	69.59	73.21	**128.07**	80.1	46.51	70.5	61.9	57.11	87.97	**104.79**	**100.07**
TypeH	FV_e	32	**33**	30	**56**	30	22	**33**	31	25	28	29	**37**
	Sum_{FS}	80.01	81.92	73.16	**148.65**	75.48	55.85	**82.38**	75.46	63.86	69.2	70.92	**91.1**
All	FV_e	82	86	80	**138**	86	62	83	78	71	84	**107**	**109**
	Sum_{FS}	210.35	224.27	203.09	**372.23**	225.49	164.89	212.23	195.81	188.68	214.75	**274.12**	**287.17**

Table 5: 12 groups of risk evaluation formulas with the same capability to representing failed test cases

Name	REFs
Group1	**Naish2** [42]
Group2	**Jaccard** [8], Anderberg [42], Sørensen-Dice [42], Dice [42], Goodman [42], M2 [42] , Naish1 [42], DStar [63]
Group3	**Tarantula** [29] , qe [35] , CBI Inc [37], Kulczynski2 [42], Ochiai [1]
Group4	**Wong2** [66] , Hamann [42], Simple Matching [42], Sokal [42], Rogers & Tanimoto [42], Hamming etc. [42], Euclid [42]
Group5	**Wong1** [66], Binary [42], Russel & Rao [42]
Group6	**Scott** [42], Rogot1 [42]
Group7	**Ample2** [42], Arithmetic Mean [42], Cohen [42], Crosstab [64]
Group8	**Wong3** [66]
Group9	**Fleiss** [42]
Group10	**GP02** [79]
Group11	**GP03** [79]
Group12	**GP19** [79]

up the fitness scores it gets on all faulty versions that satisfy the criterion of "$k == r$", as shown in Formula 11.

$$Sum_{FS} = \sum_{i}^{FV_e} FitnessScore_i \tag{11}$$

Where FV_e is the number of "$k == r$" faulty versions when using the EFF. $FitnessScore_i$ is the clustering effectiveness on the i^{th} faulty version, which is calculated by Formula 3.

4.4 Environments

We generate multi-fault versions and collect program coverage on Ubuntu 16.04.1 LTS with GCC 5.4.0. The evolution model is built upon DEAP 1.3.1 and runs on Hp Apollo 2000 equipped with 160 CPU cores with 2.4GHz and 96 GB of memory.

5 RESULTS AND ANALYSIS

As mentioned in Section 1, numerous risk evaluation formulas have been proposed in the last four decades [11], and they have been investigated theoretically or empirically by researchers. For example, Naish et al. analyzed more than 30 REFs for their capability of fault localization [42], while 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 explore [72]. In addition, Crosstab [64] and DStar [63] proposed by Wong et al., as well as GP02, GP03, and GP19 that were presented by Yoo [79] and investigated by Xie et al. [73], are also representative techniques in the field of SBFL. As a consequence, we choose the mentioned 35 REFs as baselines. We reorganize them into 12 disjoint groups, as shown in Table 5, since 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) from each group is selected for analyses because the others in the group are equivalent to it in failure representation.

We report the test results on SIR programs in Section 5.1, 5.2, and 5.3. The test results on Defects4J programs are reported in Section 5.4, as the reason given in Section 4.2.2.

5.1 The Effectiveness of Existing REFs

For each group of REFs, we invoke Formula 11 to obtain its failure representation capability on the test set, as shown in Table 4.

From the perspective of the number of faults, the groups of REFs with the highest value of FV_e in 2-bug, 3-bug, 4-bug, and 5-bug scenarios are Group4 (49), Group5 (40), Group11 (44), and Group10 (33), respectively, and the groups of REFs with the highest value of Sum_{FS} in the four scenarios are Group4 (150.16), Group5 (107.69), Group11 (105.59), and Group10 (77.56), respectively. **From the perspective of fault types,** the groups of REFs with the highest value of FV_e in TypeA, TypeP, and TypeH scenarios are Group11 (39), Group4 (46), and Group4 (56), respectively, and the groups of REFs with the highest value of Sum_{FS} in the three scenarios are Group11 (98.41), Group4 (128.07), and Group4 (148.65), respectively. **If we review the results globally** instead of breaking them down into the mentioned seven local-environments, the greatest group of REFs is Group4, with the ranking lists it produces leading the fault isolation step to precisely estimate the number of faults on 138 faulty versions and get a clustering effectiveness score of 372.23.

The second and third highest values of FV_e and Sum_{FS} are not elaborated here due to the limited space. We highlight those in Table 4 using varied shades of background colors.

5.2 The Effectiveness of Newborn EFFs

We choose a small portion of EFFs generated by the evolution framework[11] for the test, and find that many of them outperform the current 12 groups of REFs in terms of failure representation. We analyze the top-10 EFFs to stress the promise of the proposed evolution framework, whose expressions and clustering performance

[11]The time cost of evolving one generation is about 40 hours.

Table 6: The clustering effectiveness of 10 evolved fingerprinting functions on SIR

		EFF10-83	EFF7-34	EFF11-82	EFF8-137	EFF8-16	EFF11-42	EFF5-98	EFF11-8	EFF3-149	EFF4-38
2-bug	FV_e	60 (22.45%)	58 (18%)	68 (38.78%)	64 (30.61%)	27	45	23	32	45	66 (34.69%)
	Sum_{FS}	182.82 (21.75%)	180.15 (19.97%)	214.3 (42.71%)	186.88 (24.45%)	85.7	139.07	75.27	96.37	137.46	210.8 (40.38%)
3-bug	FV_e	60 (50.00%)	45 (12.50%)	48 (20.00%)	44 (10.00%)	48 (20.00%)	46 (15.00%)	37	37	47 (17.50%)	43 (7.50%)
	Sum_{FS}	154.69 (43.64%)	124.43 (15.54%)	128.43 (19.26%)	120.27 (11.68%)	123.14 (14.35%)	121.93 (13.22%)	99.85	98.53	125.11 (16.18%)	119.21 (10.70%)
4-bug	FV_e	45 (2.27%)	44	30	34	45 (2.27%)	31	61 (38.64%)	33	33	19
	Sum_{FS}	108.16 (2.43%)	104.3	71.9	82.03	111.95 (6.02%)	78.05	147.46 (39.65%)	80.59	82.63	48.55
5-bug	FV_e	43 (30.30%)	32	24	26	49 (48.48%)	33	36 (9.09%)	56 (69.70%)	26	11
	Sum_{FS}	103.05 (32.86%)	74.99	54.37	59.4	118.42 (52.68%)	81.2 (4.69%)	87.54 (12.87%)	133.92 (72.67%)	63.98	26.46
TypeA	FV_e	73 (87.18%)	62 (58.97%)	54 (38.46%)	55 (41.03%)	62 (58.97%)	43 (10.26%)	57 (46.15%)	58 (48.72%)	46 (17.95%)	39
	Sum_{FS}	191.64 (94.74%)	171.41 (74.18%)	155.58 (58.09%)	153.72 (56.20%)	162.3 (64.92%)	116.26 (18.14%)	148.79 (51.19%)	149.95 (52.37%)	127 (29.05%)	116.32 (18.20%)
TypeP	FV_e	69 (50.00%)	52 (13.04%)	57 (23.91%)	53 (15.22%)	57 (23.91%)	65 (41.30%)	57 (23.91%)	51 (10.87%)	43	46
	Sum_{FS}	179.34 (40.03%)	138.12 (7.85%)	150.89 (17.82%)	143.81 (12.29%)	148.62 (16.05%)	177.19 (38.35%)	149.79 (16.96%)	135.99 (6.18%)	118.67	134.78 (5.24%)
TypeH	FV_e	66 (17.86%)	65 (16.07%)	59 (5.36%)	60 (7.14%)	50	47	43	49	62 (10.71%)	54
	Sum_{FS}	177.74 (19.57%)	174.33 (17.28%)	162.53 (9.34%)	151.05 (1.61%)	128.29	126.8	111.54	123.47	163.51 (10.00%)	153.93 (3.55%)
All	FV_e	208 (50.72%)	179 (29.71%)	170 (23.19%)	168 (21.74%)	169 (22.46%)	155 (12.32%)	157 (13.77%)	158 (14.49%)	151 (9.42%)	139 (0.72%)
	Sum_{FS}	548.72 (47.41%)	483.86 (29.99%)	468.99 (25.99%)	448.58 (20.51%)	439.21 (17.99%)	420.25 (12.90%)	410.12 (10.18%)	409.41 (9.99%)	409.18 (9.93%)	405.02 (8.81%)

Table 7: 10 evolved fingerprinting functions

Name	Formula expression	Name	Formula expression
EFF10-83	$N_{UF}^{N_{CS}} + N_{CF} - 1$	EFF7-34	$2N_{US} + (N_S - N_{UF}) \times N_{CS}$
EFF11-82	$\lvert N_C - 1 \rvert$	EFF8-137	$N_{US}^{N_S - N_{UF} + N_{CF} + 1}$
EFF8-16	$\left\lvert 2N_{CS} + N_{US} - \frac{N_{UF}}{N_{CS}} \right\rvert$	EFF11-42	$N_{CS}^{N_S} - N_{UF}$
EFF5-98	$(N_{CS} - N_{UF}) + N_{US} \times N_{CS}$	EFF11-8	$N_{CS}(N_{UF} + N_{CS} + 1) - N_{UF}$
EFF3-149	$\frac{N_{UF}^{N_{UF}}}{N_{UF} - N_{CF} - N_{US} - 1}$	EFF4-38	$-N_{CS} \times \left(N_{CF}^2 + N_{US} \times N_{CS}\right)$

are presented in Table 7 and Table 6, respectively. The naming convention of the evolved fingerprinting functions is formalized into "EFF+*Generation-id*", for example, EFFx-y denotes the y^{th} individual of the x^{th} generation in the evolution process.

Particularly, EFF10-83 is found to be highly competitive across the evolved fingerprinting functions. It precisely predicts the number of faults on 208 faulty versions and scores 548.72 points on clustering effectiveness in the test.

5.3 Contrast Analysis

To reveal the competitiveness of EFFs generated by our evolution framework, we compare them with the corresponding best groups of REFs in both local and global environments, and accordingly set varied shades of background colors for the cells in Table 6. For example, as shown in Table 4, in the 2-bug scenario, the best of the existing 12 groups of REFs for FV_e and Sum_{FS} is Group4, which leads the fault isolation step to precisely predict the number of faults on 49 faulty versions and obtain 150.16 points on clustering effectiveness. Therefore, in Table 6, the opacity of the cells ("EFF10-83", "2-bug"-"FV_e") and ("EFF10-83", "2-bug"-"Sum_{FS}") is set to 22.45% and 21.75%, respectively, since EFF10-83 enables the number of faults to be precisely predicted on 60 faulty versions and to get 182.82 points on clustering effectiveness in the 2-bug scenario, 22.45% and 21.75%[12] higher than that of Group4, respectively.

As can be seen from Table 6, in each of seven local-environments, at least three EFFs are better than the corresponding best group of REFs regardless of FV_e or Sum_{FS}. For example, in the 2-bug scenario, EFF10-83, EFF7-34, EFF11-82, EFF8-137, and EFF4-38 dominate Group4. And in the TypeA scenario, EFF10-83, EFF7-34, EFF11-82, EFF8-137, EFF8-16, EFF11-42, EFF5-98, EFF11-8, EFF3-149, and EFF4-38 (only for Sum_{FS}) dominate Group11, and so on.

[12]Such increases are given in brackets in Table 6.

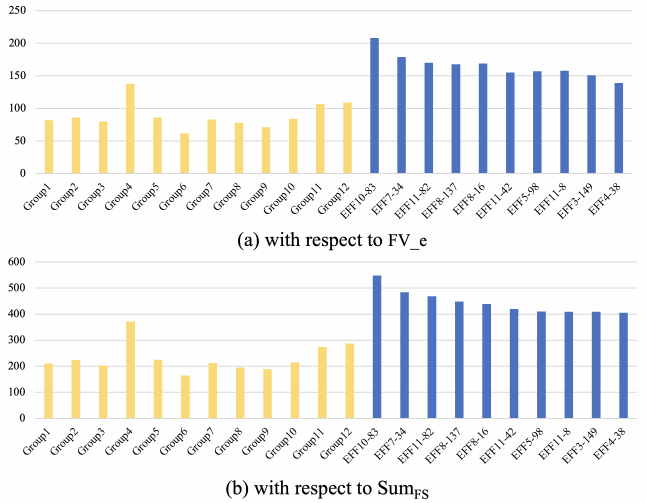

Figure 3: The contrast between existing 12 groups of REFs and 10 newborn EFFs on SIR

In a global perspective, all of the ten selected EFFs are better than the optimal existing REFs, Group4, with increases ranging from 0.72% to 50.72% for FV_e and 8.81% to 47.41% for Sum_{FS}, as shown in Table 6 and Figure 3.

These results not only demonstrate the promise of the proposed evolution framework, but also highlight a superior individual in the evolution process. As a consequence of which, for the researchers and developers who utilize SRR to represent failed test cases, we recommend employing EFF10-83 as the fingerprinting function since it 1) improves FV_e by 50.72% and Sum_{FS} by 47.41% compared with Group4 on a global scale, and 2) outperforms all 12 groups of REFs in each of local-environments.

5.4 Test in Real-world Scenarios

We also test these ten EFFs on additional 100 Java multi-fault versions. Their clustering effectiveness compared with that of 12 existing groups of REFs is given in Table 8. For real-world Java faults that deviate from the training setup (simulated C faults), the EFFs produced by our framework perform almost as well as the baselines in terms of both metrics.

Table 8: The clustering effectiveness of 12 existing groups of REFs and 10 EFFs on Defects4J

REFs	FV_e	Sum_{FS}	EFFs	FV_e	Sum_{FS}
Group1	47	186.42	**EFF10-83**	61	241.77
Group2	59	234.01	**EFF11-82**	61	242.23
Group3	56	224.00	**EFF8-16**	58	230.85
Group4	56	220.29	**EFF5-98**	57	227.21
Group5	60	238.01	**EFF3-149**	57	224.57
Group6	42	166.42	**EFF7-34**	43	170.35
Group7	47	186.42	**EFF8-137**	40	158.42
Group8	53	209.49	**EFF11-42**	62	247.21
Group9	41	162.42	**EFF11-8**	60	239.21
Group10	65	257.81	**EFF4-38**	37	146.96
Group11	64	254.37	\	\	\
Group12	61	241.40	\	\	\

We would like to emphasize that an important contribution of this paper is to provide a highly-configurable GP-based solution to automatically evolving REFs for failure representation. The selected 10 formulas in Table 7 are evolved based on the hyperparameters in Section 4.1. That is, given better hyperparameters and more rounds of evolution, our framework is possible to deliver stronger formulas than these 10.

More importantly, although the 10 evolved formulas in Table 7 do not outperform some of the baselines on real-world faults (with a slight disadvantage, can be regarded as comparable), they are highly competitive on simulated faults and are all developed by GP in an automated way. We believe that such results indicate a potential for artificial intelligence to surpass human intelligence.

6 DISCUSSION

A faulty version will be discarded if the estimated number of faults k deviates from its real number of faults r, since this work focuses on the fault isolation phase. That's to say, a series of analyses in Section 5 have nothing to do with those "$k \neq r$" faulty versions. However, such misprediction situations might reflect different deviations from r. For example, suppose that the numbers of faults of three 4-bug faulty versions are being predicted based on the ranking lists produced by two EFFs (or REFs), P_1 and P_2. We can immediately get r_i (i = 1, 2, 3) are all equal to 4. We denote $k_i^{P_1}$ and $k_i^{P_2}$ (i = 1, 2, 3) as the predicted results generated by P_1 and P_2, respectively, which are 8, 4, 1, and 5, 4, 3, respectively. Though they both make a misprediction on two faulty versions, it is visible that P_2 delivers a closer outcome, indicating that P_2 has a stronger capability in failure representation than P_1 to some extent.

We reevaluate 12 groups of REFs and 10 selected EFFs from this aspect in Figure 4, where the horizontal axis is REFs/EFFs, and the vertical axis is the value of k[13]. Three findings can be summarized from this figure: when the numbers of faults are over-predicted, no matter in what scenario, 1) Group4 is the closest group of REFs, 2) EFF10-83 is the closest EFF, and 3) EFF10-83 is closer than Group4. These three points reinforce the conclusions of Section 5.1, Section 5.2, and Section 5.3, respectively, further highlighting the potential of the proposed framework and the promise of EFF10-83.

[13]We only present the values of k when they exceed r in 2-bug, 3-bug, 4-bug, and 5-bug scenarios, analyses for the "$k < r$" situation are not given due to the limited space while are available in the public package. And also for the reason given in Section 4.2.2, this section only involves SIR programs.

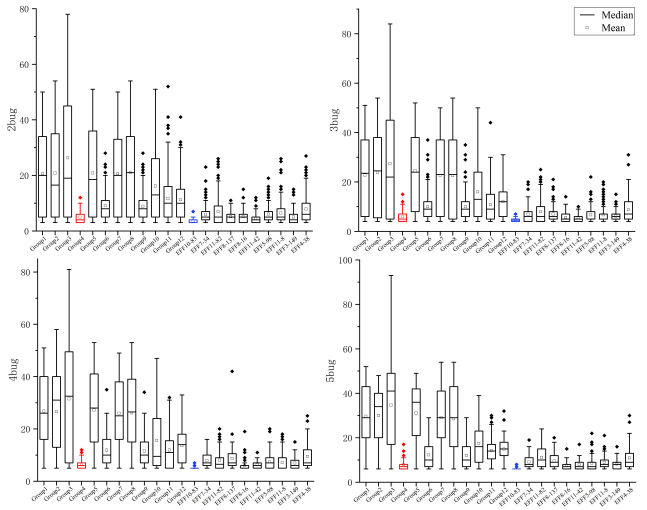

Figure 4: The values of k when they exceed r

Moreover, we also observe that in "$k > r$" scenarios, the values of k are easier to **significantly** exceed r when using the existing 12 groups of REFs, but only **slightly** exceed r when using 10 EFFs. This could be because existing REFs designed for single-fault localization are inherently unsuitable for failure representation, **as we conjectured in Section 1 and at the end of Section 2**. On the contrary, EFFs that are specifically developed for failure representation will deliver a closer prediction, thus resulting in a more cost-effective and efficient parallel debugging process.

7 THREATS TO VALIDITY

In the fitness function, we only evaluate the faulty versions that satisfy the criterion of "$k == r$", that is, if the number of faults of a program is mispredicted, it will not be sent to the following step. In parallel debugging, even if the number of faults is not precisely estimated, the subsequent localization can still be carried out. This is because if k is less than r, parallel localization can be performed iteratively, and if k exceeds r, it can be stopped when all failed test cases become successful. We do not take these two scenarios into account since the proposed framework only focuses on fault isolation, not the subsequent localization stage.

The clustering algorithm adopted in the experiments assigns one sample to one cluster, which is based on the one-failure-to-single-fault assumption. In practice, a failure might be triggered by multiple faults jointly or independently, where "jointly" involves fault interference [84], and "independently" indicates that distinct faults cause the same failure coincidently. Tackling such a one-failure-to-multiple-fault problem could be beneficial to fault isolation, even though fault isolation is usually performed with heuristic strategies.

8 RELATED WORK

The related work is organized twofold, namely, parallel debugging and failure representation.

Parallel debugging has proven to be effective and is being used by a growing community of researchers [20, 24, 28, 47, 68]. For example, Podgurski et al. recommended that developers group together

bug reports (another form of failures) with the same root cause based on supervised and unsupervised pattern classification [47]. Jones et al. adopted agglomerative hierarchical clustering to divide failed test cases, pointing out that parallel debugging can lower the time cost significantly, even if one developer handles all derived sub-tasks sequentially [28]. DiGiuseppe and Jones demonstrated that fault isolation is necessary and beneficial, despite the additional computational costs [12]. To relieve the threat posed by the mentioned one-failure-to-multiple-fault problem, Xia et al. leveraged genetic algorithm to combine 12 multi-label learning techniques [69]. Feng et al. comprehensively investigated the multi-label problem in failure clustering, which was shown to be non-trivial for the effectiveness of parallel debugging [18].

Apart from the adopted statement ranking representation and the mentioned coverage vector representation, there are also other commonly used failure proximities [41]. For example, failure point-based representation reflects the crashing venue for crashing failures, which is of great intuition but not suitable for non-crashing cases [58]. Stack trace-based representation involves an ordered list of function call sites, i.e., the place at which a function is called [10]. Predicate evaluation-based representation uniformly inserts several predicates into programs, and collects the predicate evaluations (i.e., true/false returned when a predicate is covered) during execution [40]. Dynamic slicing-based representation gathers data and control dependencies that existed among program statements, and computes the slice based on the two types of information [41]. In addition, several failure representation strategies mine other characteristics from a variety of sources. For example, focusing on semantically rich execution information, DiGiuseppe and Jones employed latent-semantic analyses to process the natural language part of source code, such as variable identifiers and comments [13]. Yoon and Yoo further extended DiGiuseppe and Jones' work. They investigated extracting features from source code at what levels could result in better clustering effectiveness [81]. Wang and Lo compared keywords in newly received and fixed bug reports, to identify whether they have the same root cause [60]. Golagha et al. presented a failure clustering scheme without coverage by extracting five predefined types of features [22]. Tian et al. introduced product difference as an extra feature, to more effectively evaluate the similarity between bug reports [55]. Fang et al. calculated the dissimilarity between test cases by comparing the relative execution frequencies of program entities [17]. To perform failure clustering when oracles are inaccessible, Tu et al. represented failures by extracting features from metamorphic slices, and the outcomes are shown to be comparable to traditional methods with oracles [57].

Delivering a risk evaluation formula exclusively for failure representation in multi-fault scenarios has not been discussed in previous works. In this paper, we aim at evolving formulas for a better failure representation capability, providing a novel insight for further improving the effectiveness of parallel debugging.

9 CONCLUSION

Focusing on how to more properly divide failures according to their root cause, in this paper, we propose an evolution framework along with a sophisticated fitness function, for automatically constructing fingerprinting functions for failure representation in multi-fault isolation. The inputs of the evolution framework involve six hyperparameters (the size of population, the maximum tree depth of fingerprinting functions, the size of selected parents, the crossover rate, copy rate, and mutation rate), one external method (fitness function), and a stopping criterion. The inputs of the fitness function involve the distance metric, the faults number estimation strategy, the medoids initialization strategy, and the clustering algorithm. The experiments on both simulated C faults and real-world Java faults demonstrate the potential of the proposed framework, and show the competitiveness of EFF10-83, the greatest fingerprinting function evolved by the framework. Last but not least, our approach is highly configurable because all of the input parameters can be changed freely, we encourage future researchers to develop better ranking-based failure proximities using our approach.

In the future, we plan to further explore the characteristic of failed executions and propose a novel type of failure proximity. A more extensive experiment with other languages and larger-scale projects is also to be considered.

ACKNOWLEDGMENTS

This work was partially supported by 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.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.
[2] Wasif Afzal and Richard Torkar. 2011. On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Systems with Applications* 38, 9 (2011), 11984–11997.
[3] Gabin An, Juyeon Yoon, and Shin Yoo. 2021. Searching for multi-fault programs in defects4j. In *International Symposium on Search Based Software Engineering*. Springer, 153–158.
[4] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering*. 402–411.
[5] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.
[6] Arun Babu, Qingkai Shi, and Muhammad Ashfaq. 2020. Python script for performing mutation testing. Github Repository. https://github.com/arun-babu/mutate.py
[7] H. L. Cao and S. J. Jiang. 2017. Multiple-Fault Localization Based on Chameleon Clustering. *Tien Tzu Hsueh Pao/Acta Electronica Sinica* 45, 2 (2017), 394–400.
[8] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 595–604.
[9] Stephen L Chiu. 1994. Fuzzy model identification based on cluster estimation. *Journal of Intelligent & fuzzy systems* 2, 3 (1994), 267–278.
[10] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1084–1093.
[11] Higor A de Souza, Marcos L Chaim, and Fabio Kon. 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347* (2016).
[12] Nicholas DiGiuseppe and James A Jones. 2011. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 international symposium on software testing and analysis*. 210–220.
[13] Nicholas DiGiuseppe and James A Jones. 2012. Concept-based failure clustering. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*. 1–4.

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

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

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

[17] Chunrong Fang, Zhenyu Chen, Kun Wu, and Zhihong Zhao. 2014. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal* 22, 2 (2014), 335–361.

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

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

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

[21] Debolina Ghosh and Jagannath Singh. 2021. Spectrum-based multi-fault localization using Chaotic Genetic Algorithm. *Information and Software Technology* 133 (2021), 106512.

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

[23] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. 2000. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability* 10, 3 (2000), 171–194.

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

[25] Y Huang and A Flynt. 2018. Exploration of Common Clustering Methods and the Behavior of Certain Agreement Indices. *Ball State Undergraduate Mathematics Exchange* 12, 1 (2018), 35–50.

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

[27] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 167–178.

[28] James A Jones, James F Bowring, and Mary Jean Harrold. 2007. Debugging in parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 16–26.

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

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

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

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

[33] Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons.

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

[35] Hua Jie Lee, Lee Naish, and Kotagiri Ramamohanarao. 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, 501–508.

[36] Yihao Li, Pan Liu, Xiao Zhao, Jiaqi Yan, and Xiaoyu Song. 2021. Evaluating the Fault-Focused Clustering Performance of Distance Metrics in Parallel Fault Localization: From an Omniscient Perspective. In *2021 7th International Symposium on System and Software Reliability (ISSSR)*. IEEE, 1–10.

[37] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *Acm Sigplan Notices* 40, 6 (2005), 15–26.

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

[39] Chao Liu and Jiawei Han. 2006. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 46–56.

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

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

[42] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[43] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. 2015. Debugging in Parallel or Sequential: An Empirical Study. *J. Softw.* 10, 5 (2015), 566–576.

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

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

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

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

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

[49] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software Engineering—Esec/Fse'97*. Springer, 432–449.

[50] SIR. 2018. *The Software Infrastructure Repository.* Retrieved November 2, 2018 from https://sir.csc.ncsu.edu/portal/index.php

[51] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

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

[53] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2016. *Introduction to data mining*. Pearson Education India.

[54] Chung Man Tang, WK Chan, Yuen Tak Yu, and Zhenyu Zhang. 2017. Accuracy graphs of spectrum-based fault localization formulas. *IEEE Transactions on Reliability* 66, 2 (2017), 403–424.

[55] Yuan Tian, Chengnian Sun, and David Lo. 2012. Improved duplicate bug report identification. In *2012 16th European conference on software maintenance and reengineering*. IEEE, 385–390.

[56] Jingxuan Tu, Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. 2019. On the analysis of spectrum based fault localization using hitting sets. *Journal of Systems and Software* 147 (2019), 106–123.

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

[58] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 612–622.

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

[60] Shaowei Wang and David Lo. 2016. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942.

[61] Shaowei Wang, David Lo, Lingxiao Jiang, Hoong Chuin Lau, et al. 2011. Search-based fault localization. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 556–559.

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

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

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

[65] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering*

42, 8 (2016), 707–740.

[66] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. IEEE, 449–456.

[67] Junjie Wu, Hui Xiong, and Jian Chen. 2009. Adapting the right measures for k-means clustering. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 877–886.

[68] Yong-Hao Wu, Zheng Li, Yong Liu, and Xiang Chen. 2020. FATOC: Bug Isolation Based Multi-Fault Localization by Using OPTICS Clustering. *Journal of Computer Science and Technology* 35, 5 (2020), 979–998.

[69] Xin Xia, Yang Feng, David Lo, Zhenyu Chen, and Xinyu Wang. 2014. Towards more accurate multi-label software behavior learning. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 134–143.

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

[71] JY XIE, Ying ZHOU, MZ WANG, and Weiliang JIANG. 2017. New criteria for evaluating the validity of clustering. *CAAI Transactions on Intelligent Systems* 12, 6 (2017), 873–882.

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

[73] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. 2013. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *International Symposium on Search Based Software Engineering*. Springer, 224–238.

[74] Xiaoyuan Xie and Baowen Xu. 2021. *Essential Spectrum-based Fault Localization*. Springer.

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

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

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

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

[79] Shin Yoo. 2012. Evolving human competitive spectra-based fault localisation techniques. In *International Symposium on Search Based Software Engineering*. Springer, 244–258.

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

[81] Juyeon Yoon and Shin Yoo. 2021. Enhancing Lexical Representation of Test Coverage for Failure Clustering. In *International Workshop on Software Engineering Automation: A Natural Language Prospective. (NLP-SEA 2021)*. 1–7.

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

[83] Abubakar Zakari and Sai Peck Lee. 2019. Parallel debugging: An investigative study. *Journal of Software: Evolution and Process* 31, 11 (2019), e2178.

[84] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* 124 (2020), 106312.

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