SURE: A Visualized Failure Indexing Approach Using Program Memory Spectrum

YI SONG, XIHAO ZHANG, and XIAOYUAN XIE, School of Computer Science, Wuhan University, Wuhan, China
SONGQIANG CHEN, The Hong Kong University of Science and Technology, Hong Kong, China
QUANMING LIU, School of Computer Science, Wuhan University, Wuhan, China
RUIZHI GAO, Sonos Inc., Boston, MA, USA

Failure indexing is a longstanding crux in software debugging, the goal of which is to automatically divide failures (e.g., failed test cases) into distinct groups according to the culprit root causes, as such multiple faults residing in a faulty program can be handled independently and simultaneously. The community of failure indexing has long been plagued by two challenges: (1) The effectiveness of division is still far from promising. Specifically, existing failure indexing techniques only employ a limited source of software runtime data, for example, code coverage, to be failure proximity and further divide them, which typically delivers unsatisfactory results. (2) The outcome can be hardly comprehensible. Specifically, a developer who receives the division result is just aware of how all failures are divided, without knowing why they should be divided the way they are. This leads to difficulties for developers to be convinced by the division result, which in turn affects the adoption of the results. To tackle these two problems, in this article, we propose **SURE**, a vi**SU**alized failu**R**e ind**E**xing approach using the program memory spectrum (PMS). We first collect the runtime memory information (i.e., variables' names and values, as well as the depth of the stack frame) at several preset breakpoints during the execution of a failed test case, and transform the gathered memory information into a human-friendly image (called PMS). Then, any pair of PMS images that serve as proxies for two failures is fed to a trained Siamese convolutional neural network, to predict the likelihood of them being triggered by the same fault. Last, a clustering algorithm is adopted to divide all failures based on the mentioned likelihood. In the experiments, we use 30% of the simulated faults to train the neural network, and use 70% of the simulated faults as well as real-world faults to test. Results demonstrate the effectiveness of SURE: It achieves 101.20% and 41.38% improvements in faults number estimation, as well as 105.20% and 35.53% improvements in clustering, compared with the state-of-the-art technique in this field, in simulated and real-world environments, respectively. Moreover, we carry out a human study to quantitatively evaluate the comprehensibility of PMS, revealing that this novel type of representation can help developers better comprehend failure indexing results.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Xihao Zhang is a co-first author.
Xiaoyuan Xie is the corresponding author.

This work was partially supported by the National Natural Science Foundation of China under the grant number 62250610224.

Authors' Contact Information: Yi Song, School of Computer Science, Wuhan University, Wuhan, China; e-mail: yisong@whu.edu.cn; Xihao Zhang, School of Computer Science, Wuhan University, Wuhan, China; e-mail: zhangxihao@whu.edu.cn; Xiaoyuan Xie (corresponding author), School of Computer Science, Wuhan University, Wuhan, China; e-mail: xxie@whu.edu.cn; Songqiang Chen, The Hong Kong University of Science and Technology, Hong Kong, China; e-mail: i9s.chen@connect.ust.hk; Quanming Liu, School of Computer Science, Wuhan University, Wuhan, China; e-mail: liuquanming@whu.edu.cn; Ruizhi Gao, Sonos Inc., Boston, MA, USA; e-mail: youtianzui.nju@gmail.com.

ACM 1557-7392/2024/12-ART210
https://doi.org/10.1145/3676958

Additional Key Words and Phrases: Failure proximity, program memory, siamese learning, failure indexing, parallel debugging

ACM Reference format:
Yi Song, Xihao Zhang, Xiaoyuan Xie, Songqiang Chen, Quanming Liu, and Ruizhi Gao. 2024. SURE: A Visualized Failure Indexing Approach Using Program Memory Spectrum. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 210 (December 2024), 43 pages.
https://doi.org/10.1145/3676958

1 Introduction

Due to the increments in the scale and complexity of modern software systems, faulty programs typically contain more than one fault [20, 26, 40, 83]. The co-existence of multiple faults has been shown to be harmful for fault localization tasks, that is, the existence of two or more faults will cause the localization of one or more of these faults to become harder [12, 20]. A mainstream way to tackle the challenge of multi-fault localization is *parallel debugging*,[1] in which a multi-fault localization task is decomposed into several sub-single-fault localization tasks by dividing[2] all failures[3] into distinct groups according to the culprit faults [18, 27, 29, 64, 75]. Such a division process is generally referred to as *failure indexing*, the goal of which is to isolate the handling of multiple faults into independent environments through the reorganization of failed test cases. Failure indexing has been proven to be diversely useful. For example, it can effectively alleviate the negative impact of multi-fault co-existence on fault localization. And for another example, the derived sub-tasks can be handled by different developers simultaneously, thus improving the efficiency of debugging [112].

For its promise, there are numerous works focusing on the topic of failure indexing [27, 34, 73]. Despite the fact that failure indexing has been integrated into many approaches or tools, it is plagued by two longstanding challenges:

— *The effectiveness of failure indexing is far from promising.* As mentioned above, the mission of failure indexing is to isolate the handling of multiple faults into separate environments by dividing failed test cases. It is obvious that an effective failure indexing process should satisfy two points: (1) The number of the derived fault-focused groups should be the same as the number of faults (i.e., *correct faults number estimation*), and (2) Failures in the same group should be triggered by the same fault, and conversely, failures in different groups should have different root causes (i.e., *promising clustering*). Many pioneering studies have pointed out that these two goals are difficult to achieve, because it is not easy to design an effective *failure proximity* (comprising the fingerprinting function and the distance metric, we will introduce them in detail in Section 2), which is the core of failure indexing [58]. The most prevalent and advanced failure proximities to date are the **code coverage (CC)**-based one and the **statistical debugging (SD)**-based one [34, 56]. However, they both rely on coverage information, which can be easily trapped and thus delivering unsatisfactory results in practice (we will further describe this point in Sections 2.2 and 3).

— *The outcome of failure indexing is hard to comprehend.* Given a collection of failed test cases, a failure indexing process will index them to their root causes and thus produce several fault-focused groups. However, developers who receive the result are only aware of how

[1]In parallel debugging, people generally consider faults that do not interfere with one another.
[2]In the current field of parallel debugging, *clustering* is typically utilized for such *division*. Thus we use these two terms interchangeably in this article.
[3]In dynamic testing, *failures* are typically *failed test cases*. In this article, we use these two terms interchangeably.

failures are divided, without knowing why they should be divided in the current manner. It is recognized that software engineering is a human-centric discipline [71], the results provided by automated debugging techniques still require being comprehended by the person who performs the following task [12, 85, 98]. In fact, nearly 30 years ago, the comprehensibility of software debugging tasks has been considered by researchers. Specifically, to make developers better comprehend fault localization results, Agrawal et al. and Jones et al. used color to visually reveal the linkage between program statements and failed/passed runs. These works have been demonstrated to be effective in helping a user localize faults [2, 40]. Parnin and Orso also realized this problem and carried out a study, explicitly indicating that human factors played a critical role in software debugging tasks [62]. Moreover, Xie et al. suggested integrating comprehension assistance into automated debugging, because the results delivered by automated debugging techniques still require being comprehended by human developers who perform the following jobs [98]. In 2023, Souza et al. continue highlighting the importance of comprehension in software debugging tasks. They employed Jaguar [67], a **spectrum-based fault localization (SBFL)** tool that visualizes fault localization results by coloring the suspicious elements from red (high) to green (low), and investigated to which extent such visualization provided hints for human developers to comprehend faulty behaviors [3]. The aforementioned representative works have garnered more than 2,700 citations. That is to say, comprehensibility has been recognized as an important topic in the field of software debugging. Seeing that failure indexing is an important class of tasks in software debugging, we can claim that *comprehension matters in failure indexing*: only when human developers comprehend the machine-produced failure indexing result, they will be convinced by the result and would like to further apply this result. Therefore, providing the failure indexing result and providing why the result is obtained are both important. However, to the best of our knowledge, existing studies only consider the former without focusing on the comprehensibility of failure indexing results.

These two challenges are worth addressing because they are in line with two problems in practice, respectively: First, the effectiveness of failure indexing determines *how good failure indexing will be*, and second, the comprehensibility of the result determines *whether the failure indexing result will be adopted by human developers.* In this article, we propose **SURE**, a vi**SU**alized failu**R**e ind**E**xing approach based on the **program memory spectrum (PMS)** for tackling the aforementioned challenges:

— *To further improve the effectiveness of failure indexing*, we propose a novel type of failure proximity, namely, the **program memory (PM)**-based failure proximity. The shortcoming of using coverage to represent failures can be partly described by the **propagation, infection, execution (PIE)** model [82], a classical theory in software testing and debugging. Specifically, the PIE model has demonstrated that a failure can be detected only if the fault[4] infects the program's internal state. However, with the coverage information only, it is very hard to explore the faulty internal state in depth during the program execution, because *being covered is a necessary but not sufficient condition for triggering a failure.* Thus, coverage cannot extract the signature of failures in deep insight. Based on this intuition, we conjecture that program internal dataflows (PM in our context) can be a finer-grained failure representation when using coverage gives rise to unsatisfactory effectiveness, because program internal dataflows can effectively embody program internal states.[5] For the fingerprinting function, the

[4]In this article, we follow the practice of general fault localization to consider the non-omission fault.
[5]This conjecture has been verified by the experiments in Section 6.

PM-based failure proximity mines and utilizes the runtime memory information (i.e., variables' names, variables' values, and the depth of stack frame, *deeper insight to explore programs' internal state than coverage*) collected during the execution of a failed test case to represent it. Specifically, we first employ SBFL techniques to determine several highly-suspicious program statements, and set these statements as breakpoints. And then, we collect the runtime memory information (i.e., variables' names and values, as well as the depth of the stack frame) at the preset breakpoints during executing a failed test case. We further reorganize the collected runtime memory information into PMS, which is in the form of an image, to serve as the proxy for a failed test case. And for the distance metric, we train a Siamese **convolutional neural network (CNN)** to predict the distance between a pair of PMS images (i.e., a pair of failed test cases), which reflects the likelihood that they are triggered by the same fault.

— *To make failure indexing results comprehensible to human developers*, we borrow the idea of Agrawal et al.'s work and Jones et al.'s work to make the failure indexing result visualized. In their work, they used color to visually map program statements in test suite executions, and help developers pinpoint faulty statements intuitively [2, 40]. In this article, we design a novel algorithm to reorganize the collected runtime memory information (which is not easy for human developers to recognize) into PMS images (which are in a human-friendly form). Specifically, for a set of runtime memory information collected during running a failed test case, we first convert variables' names and values into the numeric form according to the Ascii code of characters, and then regard variables' names, variables' values, and the depth of the stack frame as the three channels of **Red, Green, and Blue (RGB)**, a broadly-used additive color model, which will be further introduced in Section 4.3), to represent this failure in the form of PMS. As such, each failed test case is represented as a visualized image, so human developers can be easily aware of why all failures are divided the way they are when they receive the failure indexing result.

In the experiments, we obtain four C projects, Flex, Grep, Gzip, and Sed from the **Software Infrastructure Repository (SIR)** [21], and employ a mutation tool to inject simulated faults into the projects to generate faulty versions that contain one, two, three, four or five bugs (also referred to as 1-bug, 2-bug, 3-bug, 4-bug, and 5-bug faulty versions, respectively). Besides, we also use five Java projects, Chart, Closure, Lang, Math, and Time from Defects4J [42], as our benchmarks, and gather 1 ~ 5-bug real-world faulty versions according to the test case transplantation strategy proposed by An et al. [5]. In total, we generate 1,000 C simulated faulty versions and gather 100 Java real-world faulty versions. We use only 30% of the simulated faulty versions to train, and use 70% of the simulated faulty versions as well as real-world faulty versions to test. Experimental results significantly demonstrate the promise of the proposed approach: SURE can achieve 101.20% and 41.38% improvements in faults number estimation, as well as 105.20% and 35.53% improvements in clustering, compared with the state-of-the-art technique in the field, in simulated and real-world environments, respectively.

Moreover, to quantitatively measure to which extent the proposed visualized representation of failed test cases (i.e., PMS images) can be comprehensible to human developers, we conduct a human study involving 15 graduate students in Computer Science from Wuhan University and three senior programmers in top-tier Internet companies in China. The results show that compared with the most advanced and prevalent failure representation to date (i.e., the ranking-based and coverage-based ones), the representation of PMS images is more comprehensible and cost-effective: The comprehensibility of PMS images is 140.00% and 128.57% higher than the two baselines, while the time cost is only 14.57% and 14.32% of the two baselines, respectively.

The main contributions of this article are as follows:

(1) *A novel failure indexing approach*. We propose a visualized failure indexing approach, SURE, which utilizes the PMS to represent failures and uses Siamese CNNs to measure the distance between failures.
(2) *A visualized representation of failed test cases*. The PMS is a visualized representation of failed test cases, which is in a human-friendly form (i.e., images) and thus can make failure indexing outcomes more comprehensible for human developers.
(3) *A comprehensive evaluation*. We use both simulated and real-world faults in the experiments for more robust evaluation. It is worth mentioning that the training phase is performed on a small set of simulated faults while the test is conducted on a larger set of simulated faults and real-world faults. We also carry out a human study to quantitatively demonstrate the comprehensibility of the proposed PMS.
(4) *A publicly available repository*. We release the experimental data and code, as well as the material of the human study at https://github.com/SURE-Repo/SURE, to facilitate any intention of replication or reuse.

The remainder of this article is organized as follows: Section 2 introduces the background knowledge and emphasizes the motivation of this article. Section 3 provides a motivating example. Section 4 describes the technical details of SURE, and gives a running example to facilitate comprehension. Section 5 lists the research questions, datasets, evaluation metrics, and so on. Section 6 analyzes the experimental results. Section 7 discusses some interesting topics. Section 8 is about the threats to validity. Section 9 reports related works. Conclusions and directions for future work are proposed in Section 10.

2 Background

2.1 Failure Indexing

Parallel debugging is a well-recognized strategy for multi-fault localization, one of the most tedious and time-consuming problems in software testing and debugging [17, 45, 84, 92, 111]. The core of parallel debugging is to correctly divide all failures into distinct groups according to their root causes, i.e., failure indexing, thus multiple faults can be localized in isolated environments. It is obvious that failure indexing will directly determine the effectiveness of parallel debugging: the more promising failure indexing is, the better parallel debugging [38, 54, 94].

A collection of research has pointed out that failure indexing is indeed a difficult task. Failure indexing typically involves three components: A fingerprinting function that represents failures in a mathematical and structured form, a distance metric that calculates the distance between the proxies for failures, and a clustering algorithm that divides all failures into several groups based on the calculated distance information. Previous studies have made it clear that there is no clustering technique that is universally applicable in uncovering the variety of structures present in multidimensional datasets [35]. Thus, the most essential factors of failure indexing are the fingerprinting function and the distance metric (these two components are called *failure proximity*).

2.2 Fingerprinting Function

The directly available information regarding failed test cases to be indexed is twofold: the input data and the testing result (i.e., *failed*), which is too limited to effectively differentiate them. To tackle this problem, researchers typically design fingerprinting functions to mine dynamic information at runtime, so as to further extract the signature of failed test cases and use such data to represent failed test cases [16, 56, 65].

Among off-the-shelf fingerprinting functions, the CC-based strategy is the most prevalent currently [19, 32, 34], while the SD-based strategy is the most sophisticated and advanced [27, 38, 53, 73]. Here we use a faulty program containing l executable statements as an example. The CC-based fingerprinting function extracts the signature of failures from the execution path of failed test cases. It will represent failed test case f as a numerical vector of length l. There are two variants of the CC-based fingerprinting function: Cov_{hit} and Cov_{count}. As for the former, the i^{th} element of the numerical vector that represents f will be set to one (1) if f covers the i^{th} statement s_i, and zero (0) otherwise. And as for the latter, the i^{th} element of the numerical vector that represents f will be set to the execution frequency of s_i if f covers s_i, and zero (0) otherwise. The SD-based fingerprinting function extracts the signature of failures from the suspiciousness ranking list of program statements. Specifically, a failed test case f is first combined with passed test cases T_S, then the test suite $f \cup T_S$ is executed on the faulty program and the coverage is collected simultaneously. Later, an SBFL technique is employed to calculate the suspiciousness value of each program statement based on coverage information [60, 97, 103, 107], and produce a ranking list in which l program statements are descendingly ordered by their suspiciousness. As such, f can be represented as this ranking list of length l.

Despite the promise of the CC and the SD-based strategies, they share a common bottleneck: only relying on program coverage. Specifically, the intuition of the CC-based strategy is that failures triggered by the same fault should also have the same CC, and vice versa [108]. Thus, it creates numeric vectors that reflect the coverage information during the execution to represent failed test cases. The intuition of the SD-based strategy is that failures triggered by the same fault should target the same fault location, and vice versa [56, 58]. Thus, it incorporates SBFL techniques to further analyze raw coverage information, to convert it to a suspiciousness ranking list that suggests the fault location and thus can represent failed test cases. Obviously, both CC and SD purely rely on coverage information, which cannot work well if the coverage of the failures triggered by different faults is the same (we will exemplify such a situation in Section 3). That is to say, *even the most prevalent and advanced fingerprinting functions to date are still not able to deliver satisfactory results.*

2.3 Distance Metric

In failure proximity, the design of the distance metric is not an independent task, because it is tightly related to the form of the fingerprinting function. For example, the CC-based strategy represents a failed test case as a numerical vector and thus typically uses the Euclidean distance metric [81], because the Euclidean distance metric is a suitable and cheap way to handle the similarity measurement between such numerical vectors [34]. For another example, the SD-based strategy represents a failed test case as a ranking list and thus typically uses the Kendall tau distance metric [46], because the Kendall tau distance metric can properly measure how similar two ranking lists are by counting the number of pairwise disagreements between them [27]. From these, we can see that the distance metric is inseparable from the fingerprinting function: *only by designing a well-tailored distance metric can the signature of failures extracted by the fingerprinting function be fully exploited.*

3 Motivating Example

We use a motivating example in Table 1 to reveal the shortcoming of the CC-based and SD-based failure proximities mentioned above. This toy program containing 17 statements (s_1, s_2,..., s_{17}) is used to identify and replace certain words in the input string, and then output the modified string and the log message. Specifically, if an input string contains "*wordNone*" or "*wordNtwo*," these two words will be replaced with "**1**" and "**2**," respectively. The log message records the operation of

Table 1. A Motivating Example

S	Program	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
s_1	public static String process(String s){	•	•	•	•	•	•	•	•	•	•	•	•
s_2	if(s.contains("*1*") || s.contains("*2*")){	•	•	•	•	•	•	•	•	•	•	•	•
s_3	return "";}								•	•			
s_4	int sign = 0;	•	•	•	•	•	•	•			•	•	•
s_5	int sum_1 = 0;	•	•	•	•	•	•	•			•	•	•
s_6	sum_1 = s.contains("wordNone") ? 1 : 0;	•	•	•	•	•	•	•			•	•	•
s_7	sign += sum_1;	•	•	•	•	•	•	•			•	•	•
s_8	s = s.replaceAll("wordNone," "?1?"); //Fault1: "?1?" should be "*1*"	•	•	•	•	•	•	•			•	•	•
s_9	int sum_2 = 0;	•	•	•	•	•	•	•			•	•	•
s_{10}	sum_2 = s.contains("wordNtwo") ? 2 : 0;	•	•	•	•	•	•	•			•	•	•
s_{11}	sign += sum_2;	•	•	•	•	•	•	•			•	•	•
s_{12}	s = s.replaceAll("wordNtwo," "*2*");	•	•	•	•	•	•	•			•	•	•
s_{13}	if(sign == 3){	•	•	•	•	•	•	•			•	•	•
s_{14}	return "both pattern recognized";}												•
s_{15}	String msg = sign == 1 ? "wordNone recognized" : "pass";	•	•	•	•	•	•	•			•	•	
s_{16}	msg = sign > 2 ? "wordNtwo recognized" : msg; //Fault2: ">2" should be "==2"	•	•	•	•	•	•	•			•	•	
s_{17}	return s + "//" + msg;}	•	•	•	•	•	•	•			•	•	
	Result	**F**	**F**	**F**	**F**	**F**	**F**	**S**	**S**	**S**	**S**	**S**	**S**

Table 2. The Representation for $f_2 \sim f_6$

Failure Proximity	**Representation**
The CC-based	[1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
The SD-based	[1, 1, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 15, 15, 15]

the program, for example, "*wordNone recognized*," "*wordNtwo recognized*," "*both pattern recognized*," and "*pass*." In this program, statements s_8 and s_{16} each contain a fault.

Given a test suite containing 12 test cases: t_1 = "*speak wordNone*," t_2 = "*wordNone*," t_3 = "*wordNonecontained*," t_4 = "*wwwwordNoneeee*," t_5 = "*has wordNtwo*," t_6 = "*wordNtwo*," t_7 = "," t_8 = "*midd*1*le*," t_9 = "**1*2**," t_{10} = "*a normal sentence*," t_{11} = "*wordnonewordNtw*," and t_{12} = "*wordNone and wordNtwo*." The coverage information of them is given in Table 1, where a dot in the cell $[t_i, s_j]$ indicates that t_i covers s_j (i = 1, 2,..., 12, and j = 1, 2,..., 17). Among these test cases, $t_1 \sim t_6$ are failed test cases due to the inconsistency between the actual output and the expected output ($t_1 \sim t_6$ can be referred to as $f_1 \sim f_6$, respectively). More concretely, $f_1 \sim f_4$ are triggered by $Fault_1$, while f_5 and f_6 are triggered by $Fault_2$. The remaining six test cases, i.e., $t_7 \sim t_{12}$, pass the test (they can be referred to as T_S).

As mentioned in Section 2.2, the CC-based failure proximity represents a failure as a CC vector, i.e., the corresponding column in Table 1. For example, f_1 covers all program statements except s_3 and s_{14}, thus, as for Cov_{hit}, f_1 can be represented as a binary vector of length 17: [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]. By observing Table 1, we can find that the CC-based representation for the other five failed test cases ($f_2 \sim f_6$) is the same as that for f_1, as shown in the second row of Table 2. And as for Cov_{count}, the number "1" in the binary vector will be replaced with the execution frequency, which does not help to differentiate the six failures either. Thus, neither Cov_{hit} nor

Cov_{count} can effectively perform failure indexing in this example, i.e., the scenario where failures caused by different faults have the same execution coverage.

Also recall the description in Section 2.2, the SD-based failure proximity represents a failed test case as a suspiciousness ranking list of program statements based on this failed test case and passed test cases T_S. For example, to represent f_1, we can combine f_1 with T_S to form a new test suite, $f_1 \cup T_S$, run this test suite on the program, and collect the coverage information. The coverage will be delivered to an SBFL technique to calculate risk values of being faulty for each program statement, and then produce a suspiciousness ranking list by descendingly ordering statements by their risk values. Here we employ a recognized SBFL technique, GP03 [105], whose expression is given in Formula (1), to complete this process[6].

$$Suspiciousness_{GP03} = \sqrt{|N_{CF}^2 - \sqrt{N_{CS}}|}. \quad (1)$$

The generated suspiciousness ranking list is [1, 1, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 15, 15, 15][7], which will serve as the proxy for f_1. However, the SD-based representation for the other five failed test cases ($f_2 \sim f_6$) is the same as that for f_1, as shown in the third row of Table 2. That is, despite the incorporation of SBFL techniques, the SD-based failure proximity can still not differentiate the six failed test cases, because the SD-based strategy essentially relies on program coverage as well, while the six failed test cases triggered by different faults here exhibit the same execution coverage.

That is to say, when failures caused by different faults have identical coverage, even the most widespread and state-of-the-art failure proximities are still not enough to deliver promising outcomes. This situation, is, unfortunately, very common in practice, as a prestigious work in the field of failure indexing pointed out previously:

> *"A significant portion of execution profiles would be the same even if these failures are due to different faults"* [58].

Thus, developing a better failure proximity is of great significance.

Given observed failures, the essence of failure indexing tasks is to analyze the linkage among observed failures according to the underlying faults. However, observed failures are abnormal behaviors exposing at the interface of programs, while underlying faults reside at the level of code, causing a gap between failures and faults. The PIE model, a classical theory in the field of software testing and debugging, has demonstrated that a failure can be detected only if the fault infects the programs' internal state. This provides a potential perspective for alleviating the gap between observed failures and underlying faults: utilizing data from the programs' internal state to fingerprint failures. Actually, previous works have preliminarily attempted to mine the programs' internal state for failure indexing. For example, both CC and SD-based failure proximities employ coverage information as a representative of the programs' internal state. However, with the coverage information only, it is very hard to explore the faulty internal state in depth during the program execution, because being covered is a necessary but not sufficient condition for triggering a failure. Thus, coverage cannot extract the signature of failures in deep insight, it is necessary to utilize another type of data as a representative of the programs' internal state. Here we naturally think of PM information (including variables' name, variables' value, and the depth of the stack frame), because *PM information can intrinsically explore the programs' internal state, it is an intuitive*

[6]SBFL techniques typically involve four spectrum notations: N_{CF} and N_{CS} represent the number of test cases that execute the statement and return the testing result of failed or passed, 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 passed, respectively.
[7]In light of the experience of previous works [27, 34, 99], if several statements with the same suspiciousness form a tie, the rankings of all statements in the tie will be set to the beginning position of this tie.

way to understand underlying faults and thus can be one reasonable candidate to represent failures [37, 86, 102]. As a reminder, in recent years, researchers in the field of fault localization (another important sub-domain of software debugging) have been attempting to employ memory information (e.g., runtime variable information) to strengthen traditional fault localization techniques that are based on coverage information only. For example, Wen et al. claimed that "*developers often set breakpoints at specific locations to inspect if abnormal values will be observed for concerned variables, this strategy can facilitate developers in understanding the target fault*" [86]. Their work indicates that the underlying fault could be analyzed by observing runtime variables' values. Similarly, Jiang et al. stated that "*fault-relevant variables may exhibit different values in failed and passed test runs, and variables that have higher discrimination ability have a larger possibility to be the root causes of the failure*" [37]. This work demonstrates the capability of variables' values in discriminating failed and passed runs, indicating the potential of variables' values in discriminating failed runs that are triggered by different faults. Besides, Yang et al. analyzed developer behavior in software debugging, and concluded that setting breakpoints and viewing variables are important to guide developers to understand faults [102]. To summarize, program runtime memory information has the potential to be an effective source of data for fingerprinting functions in failure indexing. Based on the above intuition, in this article, *we propose a novel failure indexing approach SURE, to make the abstract potential of memory information in failure representation concrete.*

Moreover, neither the CC-based nor the SD-based failure proximity represents failures in a human-friendly form, it is hard for human developers to comprehend the failure indexing result based on such forms of failure representation. Specifically, whether a numerical vector in the CC-based strategy or a ranking list in the SD-based strategy, its length is equal to the number of executable statements of the faulty program. Unlike our motivating example, in real-world debugging, a faulty program can easily have tens of thousands of statements. As a consequence of which, the proxy for a failure will be a numerical vector or a ranking list of great length. It is obvious that in such a scenario, given failure indexing results, it is not easy for human developers to comprehend why these numerical vectors or ranking lists should be clustered into the same (or different) group(s). SURE further converts the raw memory information to PMS, which is in the form of visualized images, human developers can easily comprehend the failure indexing results given such human-friendly failure representation.

4 Approach

The PM-based failure proximity utilizes the runtime memory information to represent failures, and measures the distance between failures based on the characteristics of memory information. Based on this description, we propose our approach, SURE. The overview of SURE is depicted in Figure 1. For a faulty program under test and a given test suite containing multiple failed test cases, SURE performs the following steps to achieve failure indexing:

(1) *Breakpoint determination*. We first employ an SBFL technique to obtain the suspiciousness of all program statements, and then set the Top-x% riskiest statements as the breakpoints (the determination of the value of x will be investigated in the first research question of Section 5.1).
(2) *Collection of runtime PM*. For a failed test case, we run it on the faulty program, and collect the runtime memory information at the preset breakpoints. Specifically, the runtime memory information we collect includes variables' names, variables' values, and the depth of the stack frame.
(3) *Generation of PMS*. The collected memory information will be converted to PMS, which is in the form of visualized images in a human-friendly way.

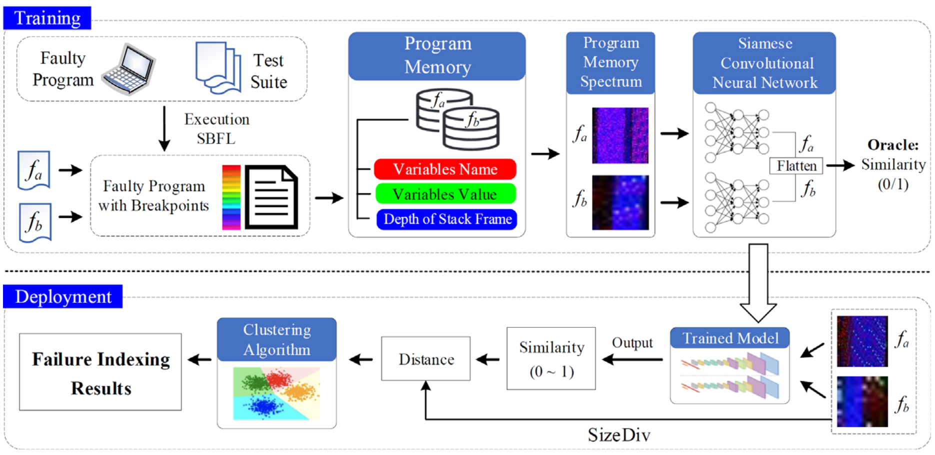


Fig. 1. Overview of SURE.

(4) *Model training*. For any pair of PMS images (i.e., any pair of failed test cases), we will assign a label to it: If these two failed test cases are triggered by the same fault, the label is one (1), indicating that the similarity between them should be one (1). Otherwise, if these two failed test cases have different root causes, the label is zero (0). Then, we feed such labeled pairs of PMS images to a Siamese CNN to train.
(5) *Prediction*. Once the model is trained, it can be used to handle the prediction of the similarity between two unseen PMS images.
(6) *Distance calculation*. For the value of the similarity between a pair of failed test cases, we first subtract it from 1, and then multiply this difference by a parameter, *SizeDiv*, as the final distance. *SizeDiv* reflects the divergence between the sizes of two PMS images.
(7) *Clustering*. The clustering algorithm will receive the distances between any pair of failed test cases, and based on which produces one or more clusters of failed test cases.

We detailedly introduce the above seven steps of SURE in Sections 4.1–4.7, respectively. Moreover, we give a running example to facilitate readers' comprehension in Section 4.8.

4.1 Breakpoint Determination

Given a faulty program P containing l statements, a test suite T comprising failed test cases T_F and passed test cases T_S ($T_F \cup T_S = T$ and $T_F \cap T_S = \varnothing$). We employ an SBFL technique to calculate the suspiciousness of l program statements of P based on T, and rank all statements in descending order of suspiciousness. Then, we determine the Top-x% riskiest statements as the breakpoints (the value of x will be investigated in the first research question of Section 5.1). As such, we can get q breakpoints, bp_1, bp_2,... bp_j,..., bp_q, where the value of q is $\lfloor l \times x\% \rfloor$, and bp_j is the breakpoint ranked j^{th} in suspiciousness.

The intuition behind this strategy is that the statements with higher risk values are more likely to be faulty, and runtime memory information gathered at these positions could have a stronger capability to reveal faults, thus can contribute more to representing failures.

As a reminder, SBFL is not required to determine the exact position of faults in this process. We simply need to identify a collection of highly-risky program statements, which are sufficient to specify breakpoints that can monitor the execution of abnormal programs.

4.2 Collection of Runtime PM

For the ith failed test case in T_F, $f_i \in T_F$, executing it on P and gathering the runtime memory information at each breakpoint. Specifically, the collected memory information of f_i is:

$$MI_i = \left\{bp'_1 : V_1^i, bp'_2 : V_2^i, ..., bp'_j : V_j^i, ..., bp'_q : V_q^i\right\}, \tag{2}$$

where V_j^i is the memory information collected at bp'_j during executing f_i. Notice the difference between bp'_j here and bp_j in Section 4.1: bp'_j is the jth breakpoint to be executed during the actual running of f_i. This arrangement integrates control flows information into our method, because program control flows are also helpful for failure representation to an extent. Supposing that there are m_{ij} variables queried at bp'_j during the running of f_i, V_j^i comprises three lists with length m_{ij}: ${V_{name}}_j^i$, ${V_{value}}_j^i$, and ${V_{depth}}_j^i$:

${V_{name}}_j^i$ contains the names of the m_{ij} variables queried at bp'_j during running f_i:

$${V_{name}}_j^i = [name_1, name_2, ..., name_{m_{ij}}], \tag{3}$$

${V_{value}}_j^i$ contains the values of these m_{ij} variables:[8]

$${V_{value}}_j^i = [value_1, value_2, ..., value_{m_{ij}}], \tag{4}$$

${V_{depth}}_j^i$ contains the depth of the stack frame of these m_{ij} variables:

$${V_{depth}}_j^i = [depth_1, depth_2, ..., depth_{m_{ij}}]. \tag{5}$$

If we are from the perspective of a failed test case, that is, considering all breakpoints together, we can integrate the memory information collected at all breakpoints into a hunk:

$$V_{name}^i = {V_{name}}_1^i \oplus {V_{name}}_2^i \oplus ... \oplus {V_{name}}_j^i \oplus ... \oplus {V_{name}}_q^i \tag{6}$$

$$V_{value}^i = {V_{value}}_1^i \oplus {V_{value}}_2^i \oplus ... \oplus {V_{value}}_j^i \oplus ... \oplus {V_{value}}_q^i \tag{7}$$

$$V_{depth}^i = {V_{depth}}_1^i \oplus {V_{depth}}_2^i \oplus ... \oplus {V_{depth}}_j^i \oplus ... \oplus {V_{depth}}_q^i \tag{8}$$

where $\oplus$ denotes the append operation between two lists. We use m_i to denote the length of V_{name}^i, V_{value}^i, and V_{depth}^i, which is the number of variables queried at all breakpoints during the running of f_i:

$$m_i = \left|V_{name}^i\right| = \left|V_{value}^i\right| = \left|V_{depth}^i\right| = m_{i1} + m_{i2} + ... + m_{ij} + ... + m_{iq}. \tag{9}$$

During the execution of a failed test case, when the program stops at a breakpoint, only the memory data at the position before the execution of that breakpoint can be collected. To ensure that memory information is gathered regarding the predetermined positions produced by SBFL, when faulty programs stop at each of the breakpoints, we continue executing a further step and then carry out the collection operation. Besides, if a program statement is executed for more than once, we collect variables' values when the execution is completed, since the latest value reflects the entire accumulation of the execution. As a reminder, for the sake of cost saving, if a program statement contains function calls, we concentrate on the original location of the preset breakpoint rather than iteratively navigating to the callee to query memory information (Experiments in Section 6 demonstrate that this strategy is good enough for failure indexing).

[8]In our experiments, we find that regarding a variable's value as a string is beneficial for distance measurement. For some special types of variables, further action will be taken. For example, for a pointer, we will further index its value by address.

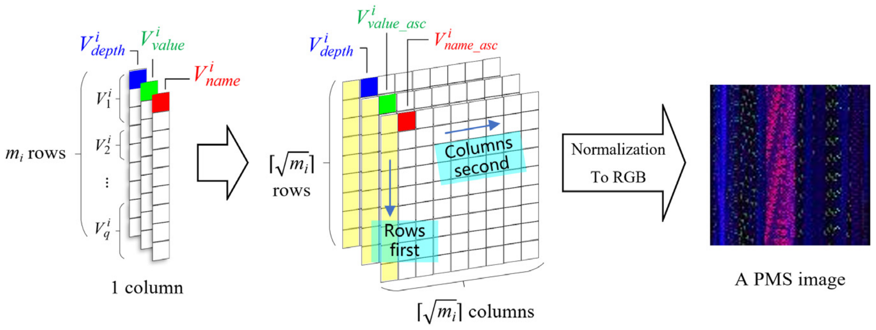


Fig. 2. Workflow of PMS images generation.

4.3 Generation of PMS

For each $f_i \in T_F$, we have collected the runtime memory information regarding it, i.e., MI_i. Now we introduce based on MI_i, how to generate the corresponding program memory spectrum PMS_i.

The workflow of the generation of PMS images is illustrated in Figure 2. Specifically, MI_i is a matrix with the shape of $(1 \times m_i \times 3)$, as shown in *the leftmost sub-figure of Figure* 2, where m_i is the number of variables queried at all breakpoints during the running of f_i, and "3" indicates the three dimensions: variables' names, variables' values, and the depth of the stack frame. The arrangement of this $(1 \times m_i \times 3)$ matrix is in the order of execution of breakpoints. Specifically, V^i_1, i.e., the runtime memory information collected at the first executed breakpoint during running f_i, is at the top. Similarly, V^i_q, i.e., the runtime memory information collected at the last executed breakpoint (there are q breakpoints in total) during running f_i, is at the bottom.

In practical development, the value of m_i is generally very large, thus the shape of this $(1 \times m_i \times 3)$ matrix is quite unbalanced. We reshape this matrix to a $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix. The principle of this reshaping process is "row first, column second." Specifically, we take the first $\lceil\sqrt{m_i}\rceil$ elements from top to bottom in the $(1 \times m_i \times 3)$ matrix as the first column of the $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix, and then take the $(\lceil\sqrt{m_i}\rceil + 1)^{\text{th}}$ to the $2\lceil\sqrt{m_i}\rceil^{\text{th}}$ elements from top to bottom in the $(1 \times m_i \times 3)$ matrix as the second column of the $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix, and so on. Notice that if m_i is not a perfect square number, few columns on the far right of the $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix cannot be filled. In such a scenario, the few remaining elements will be set to the number zero. After the reshaping process is completed, a $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix can be obtained, as shown in *the middle sub-figure of Figure* 2.

The reshaped matrix can be regarded as three $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 1)$ sub-matrices: [:, :, 0], [:, :, 1], and [:, :, 2], in which the first sub-matrix is actually V^i_{name}, and the second as well as the third sub-matrices are V^i_{value} and V^i_{depth}, respectively. Among them, the elements of V^i_{name} and V^i_{value} are in the form of characters, and the elements of V^i_{depth} are integers. To make them have a uniform format, we employ an existing method in the reference [52] to convert elements of V^i_{name} and V^i_{value} from the character form to the numeric form. Specifically, for a variable's name or a variable's value in the string form (denoted as str), we use the following formula to convert it to an integer (denoted as str_{asc}):

$$str_{asc} = \sum_{i=0}^{|str|} (i+1) * Ascii(str[i]), \tag{10}$$

where $Ascii()$ gets the Ascii code given a character (the strategy of converting strings to values by employing Ascii codes has been adopted by many previous works [59, 61, 80]). The multiplication by $(i + 1)$ is to handle the case where two strings have the same batch of characters but in different order. For example, a string "$Ee01$" can be converted to an integer 611, which is calculated by $1 * Ascii(\text{'}E\text{'}) + 2 * Ascii(\text{'}e\text{'}) + 3 * Ascii(\text{'}0\text{'}) + 4 * Ascii(\text{'}1\text{'})$.

Now, each element of the $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix is already an integer, this three-channel matrix is ready to be transformed into a colored image. Here we employ RGB, a broadly-used additive color model, to complete this transformation process, because of its prevalence, availability, and simplicity. In RGB, three primary colors each have values ranging from 0 to 255, they can be combined together in various ways to reproduce a broad array of colors in the visible spectrum [23, 31]. To make the $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix adapt to the RGB model, we normalize the elements in each of its sub-matrices, namely, [:, :, 0], [:, :, 1], and [:, :, 2] separately, according to the mentioned value range of RGB. The normalization can also alleviate the imbalance among the three dimensions, because the three sub-matrices reflect different resources of memory information thus have different scales. After normalization, the $(\lceil\sqrt{m_i}\rceil \times \lceil\sqrt{m_i}\rceil \times 3)$ matrix can be directly transformed into a colored image, as shown in *the rightmost sub-figure of Figure* 2.

One of the motivations for transforming runtime memory information into PMS images is to facilitate the following task of distance calculation. Specifically, memory information is raw data collected for failure representation, its scale could be typically large, and its form is not easy to recognize or handle by machines either. *Images are a proper candidate form to rephrase raw memory data, because the form of images can effectively integrate various sources of information of memory data into a single block, such a re-representation for raw data can provide well-organized and highly-structured objects for the downstream deep learning technique that performs distance calculation.* Moreover, the form of visualized images can also boost human developers' comprehension of failure indexing results, because images can be smoothly recognized by human beings at a glance.

4.4 Model Training

The training phase of SURE is illustrated in the upper part of Figure 1. Specifically, for a pair of failed test cases, we first collect the corresponding two sets of memory information on the faulty program with breakpoints, and then transform them into two PMS images, respectively, as mentioned in Sections 4.1–4.3.

We use the idea of Siamese learning to construct a similarity prediction model for two failed test cases (in the form of PMS images), because Siamese learning is a classical supervised learning strategy in similarity learning tasks [11, 13, 15, 72]. The goal of a Siamese learning-based model is to learn a similarity metric between pairs of input samples, it typically consists of two identical sub-networks for feature extraction that share the same structure and weights, as well as fully connected layers that are responsible for taking the learned representations from the mentioned feature extraction networks, thus producing a final output that reflects the similarity between two input samples.

The architecture of our Siamese learning model is designed as follows: two identical feature extraction networks (will be specified in the second research question of Section 5.1), and two fully connected layers (with a ReLU activation function in between). The loss function we use is the binary cross entropy loss, which is commonly used in deep learning especially for binary classification tasks, as shown in Formula (11):

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log\left(p_i\right) + \left(1 - y_i\right)\log\left(1 - p_i\right)\right], \tag{11}$$

where N is the number of the data samples, y_i is the true binary label for the i^{th} sample, and p_i is the predicted probability that the i^{th} sample is positive. As a reminder, the true binary label of a pair of PMS images is whether “1” (i.e., these two failed test cases are triggered by the same fault, thus the similarity between them should be one) or “0” (i.e., these two failed test cases have different root causes, thus the similarity between them should be zero). We denote pairs of PMS images with the label “1” as *positive samples*, while denote pairs of PMS images with the label “0” as *negative samples*.

In particular, we use only 30% of the simulated faults to train the model, and use 70% of the simulated faults as well as real-world faults to test the model (please refer to Section 5.3 for the details of the faults used in the experiments).

4.5 Prediction

Once the similarity prediction model is trained, it can be used to predict the similarity between two unseen PMS images (i.e., two failed test cases). This similarity reflects the likelihood of these two failures being triggered by the same fault. Before sending two images to the model, we first preprocess them into a uniform size. This is because handling two images with different sizes is hard for Siamese-based models, in light of the structure of the two feature extraction networks must be identical.

4.6 Distance Calculation

The output of our Siamese learning model is the similarity between two failures. Before sending the value of the produced similarity to the downstream clustering algorithm, we further process it by Formula (12), to make up for the loss caused by harmonizing the sizes of images in the previous step.

$$Distance = (1 - Similarity) \times SizeDiv. \tag{12}$$

We first explain the first half of this formula, i.e., “$1 - Similarity$.” The similarity produced by the Siamese model measures how similar two PMS images are. To adapt it for clustering algorithms, we subtract the value of the similarity between two failures from one, to reflect how dissimilar they are. Next, we explain the multiplication of the factor *SizeDiv*. As mentioned in Section 4.3, a PMS image representing the failed test case f_i is a square whose side length is $\lceil\sqrt{m_i}\rceil$, where m_i is the number of variables queried at all breakpoints during the execution of f_i. It is obvious that different f_i may correspond to different m_i, thus resulting in different side lengths of PMS images. Considering that it is hard for Siamese models to handle two images with different sizes, in the prediction phase, we preprocess two input images into the same size before feeding them to the model. But as mentioned previously, the side length of PMS images embodies the number of queried variables during running a failed test case, which can reveal the characteristics of a failure to some extent, and thus can also contribute to the distance measurement between two failed test cases. To alleviate this issue, we design *SizeDiv*, a factor that is able to quantitatively reflect the divergence between the sizes of two images. Specifically, *SizeDiv* is calculated by dividing the side length of the larger image by the side length of the smaller image.

4.7 Clustering

In Section 2.1, we have pointed out that there is no clustering technique that is universally applicable in uncovering the variety of structures present in multidimensional datasets [35]. Therefore, we employ the clustering component of MSeer [27], the state-of-the-art failure indexing technique to date, to complete the clustering phase of SURE. The clustering algorithm is twofold, namely, the faults number estimation and the clustering. The former aims to predict the number of clusters (i.e., the number of faults) given the distance information between data samples (i.e., failed test cases), and the latter is responsible for clustering data samples (i.e., failed test cases) into one or more groups. Next, we give a concise description of these two steps, for more details please refer to the reference [27].

For the faults number estimation. It is well-recognized that one of the trickiest challenges in clustering lies in the estimation of the number of clusters [25, 48, 78]. Putting it into the context of failure indexing, we can claim that predicting the number of faults given a number of failures is very important. The adopted clustering algorithm presented a novel mountain method-based approach inspired by the previous works [14, 101], to perform the faults number estimation and the assignment of initial medoids to these clusters at the same time. Specifically, the adopted clustering algorithm first calculates a potential value for each failed test case according to the density of its surroundings, such a potential value is used to measure the possibility of a data point being set as a medoid. And then, (1) The failure with the highest potential value will be chosen as the current medoid. (2) The potential values of all failed test cases will then be updated in accordance with their distance from the newest medoid. (3) Repeating the above two steps iteratively, until the maximum potential value falls within a certain threshold.

For the clustering. Once the number of clusters and the initial medoids are determined, all failures are ready to be clustered. The adopted clustering algorithm utilizes K-medoids, a widely-used clustering strategy, to complete this process. The K-medoids strategy sets actual (not virtual) data points as medoid and thus can be more applicable to SURE, because the mean of memory information is difficult to define. Moreover, the K-medoids strategy has been shown to be very robust to the existence of noise or outliers [44].

4.8 Running Example

In the motivating example in Section 3, we use a toy program to show that the most prevalent and advanced approach to date can still not good enough to produce satisfactory failure indexing results. That toy program successfully reveals the bottleneck of existing approaches with only 17 program statements. But such a small-scale program is not suitable to illustrate the workflow of SURE, because a small number of program statements will correspondingly result in a small number of breakpoints, thus further resulting in a small number of queried variables. In such cases, the generated PMS images will be very small, which does not affect the running of SURE but can hinder readers' comprehension as a running example. Therefore, here we use a complete running example to illustrate the workflow of SURE. It is important to mention that neither the CC nor the SD tactic can work well on this running example, that is to say, this running example can also serve as a motivating example to reveal the shortcoming of existing failure proximities. The reason why we don't do this is just to maintain the simplicity of the motivating example for readers' comprehension.

We employ Grep, a classical tool aiming to print lines in specific files that contain a match of the given patterns. We obtain Grep (v2.4) which contains 13,274 lines from SIR [70], and randomly mutate three positions of the original clean program to inject three faults into the program, to obtain a 3-bug faulty version. The three mutants are shown in Listing 1.

We compile the mutated program and run the accompanying test suite, observing a series of failures and also getting a number of passed test cases. Then, we conduct failure indexing following the aforementioned steps: For the breakpoint determination (Section 4.1), we employ an SBFL technique (e.g., DStar [89]) to determine the suspiciousness of program statements and setting breakpoints. For the collection of runtime PM (Section 4.2), we run failed test cases and collect the runtime memory information *MI* in accordance with Formula (2). *MI* comprises variables' names, variables' values, and the depth of the stack frame, whose content and structure are defined as Formulas (3)–(8), and *MI*'s scale is determined by Formula (9). For the generation of PMS images

```
3164     if (leftoversf)
3165       {
3166           copyset(leftovers, labels[ngrps]);
3167           copyset(intersect, labels[j]);
3168 -         MALLOC(grps[ngrps].elems, position, d->nleaves);
     +         MALLOC(grps[ngrps].elems, position, d->nleaves*-1);
3169           copy(&grps[j], &grps[ngrps]);
3170           ++ngrps;
3171       }
         ...
4681     for (ep = text + size - 11 * len;;)
4682       {
4683           while (tp <= ep)
4684               {
4685 -                 d = d1[U(tp[-1])], tp += d;
     +                 d = d1[U (! tp[-1])], tp += d;
4686                   d = d1[U(tp[-1])], tp += d;
4687                   if (d == 0)
4688                     goto found;
          ...
7480      compile_stack.stack = TALLOC (INIT_COMPILE_STACK_SIZE, compile_stack_elt_t);
7481      if (compile_stack.stack == NULL)
7482        return REG_ESPACE;
7484 -    compile_stack.size = INIT_COMPILE_STACK_SIZE;
     +    compile_stack.size = 0;
7485      compile_stack.avail = 0;
```

Listing 1. Mutants in the running example.

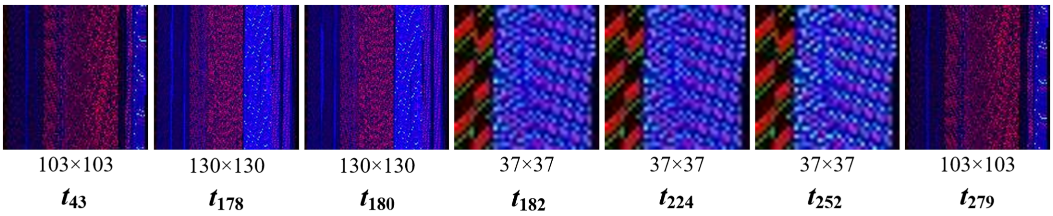


Fig. 3. The PMS images of the failed test cases.

(Section 4.3), we first convert variables' names and values to the numeric form using Formula (10), and then transform each set of memory information into a PMS image.

It is hard and not necessary to completely illustrate this running example because there are too many failed test cases. Here we randomly select seven of them, namely, t_{43}, t_{178}, t_{180}, t_{182}, t_{224}, t_{252}, and t_{279}, to show their PMS images and describe the following steps.[9] Among them, t_{43} and t_{279} are triggered by the mutant located in Line 3168, t_{178} and t_{180} are triggered by the mutant located in Line 4685, and t_{182}, t_{224} as well as t_{252} are triggered by the mutant located in Line 7484. The seven PMS images as well as their size, are given in Figure 3. Then, we employ the trained model (Section 4.4) to predict the similarity between pairs of failures (Section 4.5), and get the distance information (Section 4.6), as given in Table 3 (gray-shading cells represent other failures that are closest to a certain failure). Finally, all failures are clustered into different groups using the two phases of the clustering algorithm: the faults number estimation and the clustering (Section 4.7). The outcome of the clustering algorithm is: $\{t_{43}, t_{279}\}$, $\{t_{178}, t_{180}\}$, and $\{t_{182}, t_{224}, t_{252}\}$, which is consistent with

[9]The remaining failures omitted here can also support the conclusion of this example.

Table 3. Distance Information between Failures

	t_{43}	t_{178}	t_{180}	t_{182}	t_{224}	t_{252}	t_{279}
t_{43}	0.00	1.26	1.26	2.78	2.78	2.78	0.01
t_{178}	1.26	0.00	0.02	3.51	3.51	3.51	1.26
t_{180}	1.26	0.02	0.00	3.51	3.51	3.51	1.26
t_{182}	2.78	3.51	3.51	0.00	0.01	0.02	2.78
t_{224}	2.78	3.51	3.51	0.01	0.00	0.02	2.78
t_{252}	2.78	3.51	3.51	0.02	0.02	0.00	2.78
t_{279}	0.01	1.26	1.26	2.78	2.78	2.78	0.00

the characteristic of the distance information given in Table 3. Obviously, this clustering outcome achieves promising failure indexing since the number of clusters (i.e., the number of faults, three) is correctly estimated, and all failures are grouped correctly according to the root cause.

Moreover, we can observe that the characteristics of the PMS images in Figure 3 are highly identical to the failure indexing result. Specifically, the PMS images of t_{43} and t_{279} are similar and have the same size. Likewise, we can observe the same phenomenon on $\{t_{178}, t_{180}\}$ and $\{t_{182}, t_{224}, t_{252}\}$. With the support of the PMS images, developers can be easily convinced by the failure indexing outcome and thus adopt it.

5 Experimental Setup

5.1 Research Question

—*RQ1: Does the value of the breakpoint determination threshold impact SURE's effectiveness?*

As mentioned in Section 4.1, SURE will take the Top-x% riskiest program statements as breakpoints. We investigate how the value of x impacts the effectiveness of SURE. Specifically, we compare the effectiveness of SURE when such a threshold is set from 0% to 100% with 10% increments, i.e., 10%, 20%, 30%,..., 100%.

—*RQ2: Does the selection of the feature extraction network impact SURE's effectiveness?*

For a pair of failures, SURE uses a Siamese learning-based model to predict its distance. Because the input samples in our context are in the form of PMS images, the feature extraction network is correspondingly CNNs. In this RQ, we investigate how the feature extraction network impacts the effectiveness of SURE. Specifically, we compare the effectiveness of SURE when the network configures VGG-16 [69], AlexNet [49], and ResNet-18 [30]. The three investigated networks are all classical and commonly-used. Specifically, VGG-16 was introduced by the **Visual Geometry Group (VGG)** from the University of Oxford in 2014 [69], it is part of the VGG network series and is named "16" because it consists of 16 layers. VGG-16 is known for its simple yet effective architecture, which allows it to learn rich and complex features from images. Also, VGG-16's parameter count is relatively large thus it requires more computational resources. AlexNet is a pioneering deep CNN introduced by Krizhevsky et al. in 2012 [49], which marks a significant breakthrough in computer vision. AlexNet popularized the use of ReLU activation functions, which helps alleviate the vanishing gradient problem and accelerate training. ResNet-18 is a variant of the residual neural network (ResNet) architecture. It was introduced as part of the ResNet family by He et al. in 2015 [30], which uses residual blocks to address the vanishing gradient problem that often occurs in deep neural networks.

—*RQ3: How does SURE perform compared with the most prevalent and advanced failure indexing technique?*

In Section 2, we have introduced that the CC-based strategy has been extensively used by the failure indexing community due to its simplicity, and the SD-based strategy has been recognized as the state-of-the-art solution. Thus, we compare SURE with these two types of techniques for more convincing evaluation.

For the CC-based strategy, we select Cov_{hit} [32, 34] as our baseline, because it is the most common technique in the CC class. Specifically, it represents a failed test case as a binary numeric vector with a length equal to the number of program statements, the i^{th} element of the vector will be set to one (1) if the i^{th} statement is covered by this failed test case, and zero (0) otherwise. Considering that there are many candidate distance metrics for handling binary vectors, we try out some representative ones (e.g., the Euclidean distance, the Jaccard distance, and the Cosine distance) to make sure the baseline comparison is strong. Therefore, the baseline techniques of Cov_{hit} include $Cov_{hit\text{-}Eucli}$, $Cov_{hit\text{-}Jacc}$, and $Cov_{hit\text{-}Cos}$. Moreover, seeing that there have emerged some works concerning the impact of the execution frequency of program statements on debugging, we also adopt another technique in the CC class, Cov_{count} [68, 79, 87], as our baseline. Specifically, Cov_{count} also represents a failed test case as a numeric vector with a length equal to the number of program statements, while the i^{th} element of the vector will be set to the actual execution frequency if the i^{th} statement is covered (rather than a constant, one), and zero otherwise.

For the SD-based strategy, we select *MSeer* as our baseline, because it is the state-of-the-art technique not only in the SD class but also in the current field of failure indexing. Specifically, it first runs a failed test case along with passed test cases against the faulty program, and inputs the gathered coverage information to Crosstab [91], a risk evaluation formula designed for fault localization, to calculate the suspiciousness of being faulty for each program statement. And then, ranking all statements in descending order according to their suspiciousness values. Such a ranking list will serve as the proxy for this failed test case. Besides, in our previous work, we found that employing risk evaluation formulas designed for fault localization to represent failures could be improper [73], and thus designed a genetic programming-based approach to automatically generate risk evaluation formulas that are solely for failure representation. According to our result, if Crosstab is replaced with EFF10-83 (the optimal risk evaluation formula evolved by our genetic programming-based approach [74]), MSeer will do much better. Thus, we manually upgrade MSeer to $MSeer_{EFF\text{10-83}}$, which employs EFF10-83 to calculate suspiciousness, and include it as our baseline.[10]

To summarize, we compare SURE with six techniques, i.e., $Cov_{hit\text{-}Eucli}$, $Cov_{hit\text{-}Jacc}$, $Cov_{hit\text{-}Cos}$, and Cov_{count} (the CC-based strategy), as well as *MSeer* and $MSeer_{EFF\text{10-83}}$ (the SD-based strategy).

—*RQ4: To what extent can SURE help human developers comprehend failure indexing results?*

As a visualized failure indexing technique, SURE represents failed test cases as human-friendly PMS images to boost the comprehension to failure indexing results. In this RQ, we quantitatively investigate to what extent developers can comprehend and further adopt the failure indexing result with the support of PMS images. Similar to RQ3, we compare the comprehensibility of SURE with that of CC-based and SD-based techniques. Specifically, we conduct a comparison of developers' performance on comprehending failure indexing results among given PMS images (i.e., the fingerprinting function of SURE), CC vectors (i.e., the fingerprinting function of CC-based strategies), and suspiciousness ranking lists (i.e., the fingerprinting function of SD-based strategies).

[10] As a reminder, $MSeer_{EFF\text{10-83}}$ is not an existing published failure indexing technique, it is manually created by us to further evaluate the competitiveness of SURE.

This human study involves nine tasks and 18 experienced developers from both Wuhan University and top-tier Internet companies.

5.2 Parameter Setting

SURE needs to first determine the suspiciousness value of program statements, and thus sets breakpoints at those highly-risky positions. In this phase, we use DStar [89], the state-of-the-art SBFL technique that has been utilized broadly. Considering the preference for DStar in many other studies (such as the references [8, 63, 88]), we set the value of * in DStar to 2, the most thoroughly-explored value, in our experiments. Such a choice is not hard-coded but can be configurable, any other fault localization techniques working at the granularity of statements can be adapted to this phase.

In the training phase, we set the value of batch size as 16. The initial value of the learning rate is defined as 1e-4, and it will be multiplied by 0.96 after each epoch.

5.3 Dataset

For comprehensive and robust evaluation, in the experiments, we use the benchmark involving both simulated faults and real-world faults. For simulated benchmarks, we manually seed different types of faults into clean programs, to obtain simulated faulty programs, in light of the fact that previous research has confirmed that mutation-based faults can provide credible results for experiments in software testing and debugging [6, 7, 22, 43, 55, 66]. The clean programs are obtained from the SIR [21]. For real-world benchmarks, we search for the Defects4J projects [42], one of the most popular datasets in the current field of software testing and debugging, according to the test case transplantation strategy proposed by An et al. [5]. All faulty programs contain one, two, three, four, or five bugs (also referred to as 1-bug, 2-bug, 3-bug, 4-bug, and 5-bug faulty versions, respectively), because such numbers of faults are most-widely investigated in previous studies [27, 51, 95, 110].

5.3.1 Simulated Faulty Programs. SIR is a classical repository in the community of software testing and debugging, which has been employed in numerous pioneering studies [33, 47, 90, 106, 111]. From SIR we obtain Flex, Grep, Gzip, and Sed, four projects that have been extensively adopted in earlier works [10, 28, 74], as the clean programs to mutate (i.e., seed faults). The concise information about them can be found in Table 4. We utilize an open-source tool with "13" fork and "23" star on GitHub to perform mutation, which defines 67 types of points that can be mutated, and provides several mutation operators for each [9]. The mutation operators we leverage can be categorized into the following two classes:

— *Assignment Fault* [36]: Editing the value of constants in the statement, or replacing the operators such as addition, subtraction, multiplication, division, etc. with each other.
— *Predicate Fault* [100]: Reversing the *if-else* predicate, or deleting the *else* statement, or modifying the decision condition, and so on.

To create an r-bug faulty version (r = 2, 3, 4, 5), the faults from r individual 1-bug faulty versions are injected into the same program. Such a strategy has been adopted by the majority of the published studies in the field of multi-fault debugging [1, 24, 32, 50, 109]. In our evaluation, 2-bug, 3-bug, 4-bug, and 5-bug faulty versions each have 240. As a reminder, we also create 40 1-bug faulty versions, because as the start point of failure indexing, 1-bug scenarios are necessary to be included in the evaluation of failure indexing techniques. The reason for a smaller scale of 1-bug faulty versions is that single-fault scenarios are not our focus. Specifically, (1) Failure indexing is designed to tackle the challenge induced by the co-existence of multiple faults, it should be evaluated on more multi-fault programs naturally (this choice has been adopted by many prior works [51, 95, 110]); and (2) The evaluation of failure indexing techniques could be confused if there

Table 4. Benchmarks

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

are too many 1-bug faulty versions, because the capability of 1-bug faulty versions to measure failure indexing techniques' effectiveness is a little bit limited. In a 1-bug faulty version, all failures are linked to the same fault and thus they should be clustered into one group, such simple cases will affect the whole evaluation, and further reduce the capability of evaluation results to essentially reveal failure indexing techniques' effectiveness. To summarize, we obtain 1,000 SIR faulty versions containing 1 $\sim$ 5 faults.

5.3.2 Real-World Faulty Programs. Defects4J is one of the most popular benchmarks in the current field of software testing and debugging, due to its realism and ease of use [42]. Defects4J is typically for single-fault scenarios, namely, no matter how many bugs are contained in a faulty program, the provided test suite is only sufficient to reveal one of them. Such a characteristic hinders its use in failure indexing studies. To adapt Defects4J to multi-fault scenarios, An et al. presented a test case transplantation strategy [5]. Specifically, the majority of Defects4J faulty versions are indexed chronologically according to the revision date, a lower ID indicates a more recent version. Therefore, the fault in a newer version is also likely to be contained in an older version. For example, the fault of the faulty version Math-5b is found to exist in the faulty version Math-6b as well, it is not revealed in Math-6b simply due to the absence of the fault-revealing test case. If this test case is transplanted to Math-6b, the enhanced test suite is able to reveal both of the two faults, thus a 2-bug faulty version can be obtained [5]. Following this strategy, we search for a collection of multi-fault Defects4J programs, from Chart, Closure, Lang, Math, and Time, as shown in Table 4. Because Defects4J multi-fault versions are obtained by searching in real-life environments, their number is not very large. In total, we obtain 100 Defects4J faulty versions containing 1 $\sim$ 5 faults. As a reminder, 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 1-bug, 2-bug, 3-bug, 4-bug, and 5-bug Defects4J faulty versions are unbalanced. Thus, the experimental results of Defects4J will be presented as a whole.

5.4 Metric

The mission of a failure indexing technique is to identify the mutual relationship among failures by clustering, i.e., determine which failures are triggered by the same (or different) fault(s). Correspondingly, the outcome of failure indexing is one or more clusters of failed test cases. For evaluation, we need to quantitatively measure to which extent the delivered one or more clusters correctly reflect the true relationship among failures. There are two typical types of metrics in evaluating clustering outcomes, namely, external metrics [93] and internal metrics [76]. The former compares

Table 5. Four Scenarios in the Pair of Test Cases-Based Metrics

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

clustering results with the oracle (true linkages between failures and faults), while the latter examines inherent properties of the delivered clusters, such as compactness and separation, and so on, typically when the oracle is inaccessible. In our controlled experiments, the oracle clusters can be obtained easily in advance. Specifically, for an r-bug SIR faulty version, it is generated by combining r individual single-bug faulty versions. We can run the test suite against these r faulty versions separately, thus becoming aware of the culprit fault of each failure. And for an r-bug Defects4J faulty version, it is generated by transplanting the failed test cases triggered by a fault to the faulty version that contains another fault. This transplantation process itself explicitly indicates the relationship between failures and underlying faults. Thus, we use external metrics to measure the effectiveness of failure indexing techniques. Concretely speaking, we employ four external metrics, the **Fowlkes and Mallows Index (FMI)**, the **Jaccard Coefficient (JC)**, the **Precision Rate (PR)**, and the **Recall Rate (RR)**, in our experiments, because they are all classical metrics for clustering and have been adopted by many published studies [93, 96, 113]. Among them, FMI and JC are pair of test cases-based, while PR and RR are single test case-based.

5.4.1 Clustering Effectiveness—Pair of Test Cases-Based Metrics. The pair of test cases-based metric refers to comparing the indexing consistency of each pair of failed test cases in the generated cluster with the oracle cluster. Four possible scenarios in the comparison are given in Table 5. Specifically, supposing that there are n failed test cases that need to be clustered, they can form C_n^2 pairs by combining any two ones. For a pair, if its two member failures are determined to be triggered by the *Same* fault in the generated cluster, and these two are also divided into the *Same* group in the oracle cluster, this pair falls into the category of "*SS*" (*Same*, *Same*). Similarly, a pair is also possible to fall into the categories of "*SD*" (*Same*, *Different*), "*DS*" (*Different*, *Same*) or "*DD*" (*Different*, *Different*). We use X_{SS}, X_{SD}, X_{DS}, and X_{DD} to denote the numbers of the pairs fallen into "*SS*," "*SD*," "*DS*," and "*DD*," respectively. Obviously, the sum of X_{SS}, X_{SD}, X_{DS}, and X_{DD} will be C_n^2, because any pair of failures will belong to one of the four scenarios. FMI and JC can integrate these four notations into one metric, to comprehensively reflect the similarity between the generated cluster and the oracle cluster, as shown in Formulas (13)–(14), respectively.

$$FMI = \sqrt{\frac{X_{SS}}{X_{SS} + X_{SD}} \times \frac{X_{SS}}{X_{SS} + X_{DS}}}, \tag{13}$$

$$JC = \frac{X_{SS}}{X_{SS} + X_{SD} + X_{DS}}. \tag{14}$$

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

5.4.2 Clustering Effectiveness—Single Test Case-Based Metrics. The single test case-based metric refers to comparing the classification result of each failed test case in the generated cluster with the oracle cluster. Four possible scenarios in the comparison are given in Table 6. Specifically, for a

Table 6. Four Scenarios in the Single Test Case-Based Metrics

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

failed test case, if it is predicted to be triggered by a fault in the generated cluster (i.e., *Positive*), and its root cause is indeed this fault (i.e., *True* prediction result), this failure falls into the category of "*TP*" (*True Positive).* Likewise, if a failed test case is predicted not to be triggered by a fault in the generated cluster (i.e., *Negative*), while its root cause is actually this fault (i.e., *False* prediction result), this failure falls into the category of "*FN*" (*False Negative*). Similarly, a failed test case is also possible to fall into the categories of "*FP*" (*False Positive*) and "*TN*" (*True Negative*). We use X_{TP}, X_{FN}, X_{FP}, and X_{TN} to denote the numbers of the failed test cases fallen into "*TP*," "*FN*," "*FP*," and "*TN*," respectively. Obviously, the sum of X_{TP}, X_{FN}, X_{FP}, and X_{TN} will be the number of failures, because any failure will belong to one of the four scenarios. PR and RR can integrate these four notations into one metric, to comprehensively reflect the similarity between the generated cluster and the oracle cluster, as shown in Formulas (15) and (16), respectively.

$$PR = \frac{X_{TP}}{X_{TP} + X_{FP}}, \tag{15}$$

$$RR = \frac{X_{TP}}{X_{TP} + X_{FN}}. \tag{16}$$

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.

Notice that FMI, JC, PR, and RR evaluate the effectiveness of clustering of one faulty version. In our experiments, there are a series of faulty versions to make our evaluation abundant. Thus, for a failure indexing technique T, we determine its effectiveness by adapting the FMI, JC, PR, and RR metrics on one faulty version to that on multiple faulty versions:

$$S\_M_M^T = \sum_{i}^{V_{equal}^T} M_i, \tag{17}$$

where "S_M" is the abbreviation for "*Sum_Metrics*." V_{equal}^T is the number of faulty versions whose number of faults is correctly predicted by using T. M_i is the metric value M (M takes FMI, JC, PR, or RR) on the i^{th} "$k == r$" faulty version[11]. As a reminder, an r-bug faulty version contains r faults, which is hard to know in advance. A failure indexing technique should first correctly predict the number of faults r, and then cluster all failures into r groups. For an r-bug faulty version, if the predicted number of faults k is not equal to r, we do not evaluate the clustering effectiveness on this faulty version, because prior studies have pointed out that the performance of a failure indexing technique can be mainly determined by those "$k == r$" faulty versions [27, 73], that is, the contribution of "k != r" ones is marginal. And also, it is indeed difficult to compare k delivered clusters with r oracle clusters. Notice that "$k == r$" is just an ideal scenario (not necessary) for SURE. Even if k != r, SURE can also work (We will further explain this point in Section 8).

[11]For an r-bug faulty version, if the predicted number of faults k is equal to r, we label this faulty version as "$k == r$".

5.4.3 ***Average Precision (AP).*** Apart from evaluating the clustering effectiveness of failure indexing techniques on those "$k == r$" faulty versions, we also intend to deeply assess the effectiveness of PMS images in measuring the proximity of failures independently of the clustering algorithm (i.e., on both "$k == r$" and "$k != r$" faulty versions). More concretely, we wonder when failures are represented as PMS images, whether failure pairs originating from the same root cause exhibit higher similarity compared to pairs from different root causes. Thus, we utilize another metric, AP, which does not rely on a precise prediction of the number of faults but focuses on the similarity between failure pairs. AP is the weighted mean of precision values at each threshold where the weight value is set to the increase in recall from the previous threshold, as defined in Formula (18):

$$AP = \sum_{n} (R_n - R_{n-1}) P_n, \tag{18}$$

where P_n and R_n are the precision and recall values at the n^{th} threshold, respectively. The baseline for AP is the proportion of positive samples.

In summary, there are six evaluation metrics used in the experiments. Among them, the first five metrics correspond to the two goals of failure indexing mentioned previously: V^T_{equal} reflects the goal of *correct faults number estimation*, and $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$ reflect the goal of *promising clustering*. The last one metric AP provides an evaluation that is less dependent on the accuracy of fault number estimation.

5.5 Environment

We collect program coverage and runtime memory information on Ubuntu 16.04.1 LTS with GCC 5.4.0 and JDB 1.8. The Siamese neural network model is trained and deployed on 4 GPUs of GeForce RTX 2080 Ti. The clustering process runs on a server equipped with 96 Intel Xeon(R) Gold 5218 CPU cores with 2.30GHz and 160 GB of memory.

6 Result and Analysis

6.1 RQ1: The Value of the Breakpoint Determination Threshold

As the first step, SURE needs to determine the Top-x% riskiest program statements as breakpoints. In this RQ, we investigate the effectiveness of SURE when the breakpoint determination threshold is set from 0% to 100% with 10% increments. Notice that in the training and deployment phases of SURE, feature extraction networks are necessary. Though RQ2 will investigate the selection of such networks, we must adopt a feature extraction network in this RQ to make the model run. To avoid bias, we will investigate this research question based on three feature extraction networks, i.e., VGG-16, AlexNet, and ResNet-18, separately. The results are given in Table 7 (in simulated environments) and Table 8 (in real-world environments). In these two tables, each bold cell denotes the value of a specific metric when using a variant of SURE, and each non-bold cell denotes the value of the variance (Var) or standard deviation ($S.D.$).

6.1.1 In Simulated Environments. Let us first focus on the impact of the breakpoint determination threshold on the effectiveness of SURE, based on VGG-16 (i.e., the row "VGG-16" of Table 7). For example, the cell ("VGG-16," "10%," V^T_{equal}) is 161, indicating that when SURE takes the Top-10% riskiest statements as breakpoints and adopts VGG-16 as the feature extraction network (SURE in such a configuration can be referred to as $SURE^{10\%}_{VGG\text{-}16}$, similarly below), it can correctly predict the number of faults on 161 faulty versions. Similarly, when T takes $SURE^{10\%}_{VGG\text{-}16}$, the values of $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$ are 126.78, 106.39, 120.24, and 80.67, respectively. We can find that when we use VGG-16 as the feature extraction network, it does not matter much if the breakpoint

Table 7. Performance of SURE on SIR

		10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
VGG-16	V^T_{equal}	**161**	**147**	**122**	**143**	**153**	**164**	**142**	**145**	**151**	**155**
	$S\_M^T_{FMI}$	**126.78**	**114.32**	**96.11**	**114.36**	**121.28**	**131.10**	**112.59**	**112.26**	**118.91**	**122.75**
	-Var	7.22e-03	6.99e-03	6.06e-03	6.98e-03	7.08e-03	5.67e-03	5.58e-03	4.72e-03	6.60e-03	5.59e-03
	-$S.D.$	8.49e-02	8.36e-02	7.78e-02	8.35e-02	8.41e-02	7.53e-02	7.47e-02	6.87e-02	8.12e-02	7.48e-02
	$S\_M^T_{JC}$	**106.39**	**94.79**	**80.36**	**96.97**	**102.25**	**111.44**	**94.95**	**93.18**	**99.82**	**103.33**
	-Var	1.55e-02	1.39e-02	1.23e-02	1.49e-02	1.49e-02	1.22e-02	1.15e-02	9.73e-03	1.33e-02	1.15e-02
	-$S.D.$	1.24e-01	1.18e-01	1.11e-01	1.22e-01	1.22e-01	1.11e-01	1.07e-01	9.86e-02	1.15e-01	1.07e-01
	$S\_M^T_{PR}$	**120.24**	**106.67**	**91.60**	**111.40**	**115.48**	**127.75**	**108.06**	**110.88**	**119.31**	**123.99**
	-Var	2.36e-02	2.37e-02	2.82e-02	2.65e-02	2.75e-02	2.46e-02	2.00e-02	1.62e-02	2.31e-02	1.65e-02
	-$S.D.$	1.54e-01	1.54e-01	1.68e-01	1.63e-01	1.66e-01	1.57e-01	1.41e-01	1.27e-01	1.52e-01	1.29e-01
	$S\_M^T_{RR}$	**80.67**	**71.20**	**59.52**	**72.80**	**75.92**	**84.55**	**69.20**	**63.20**	**72.66**	**73.17**
	-Var	2.32e-02	1.51e-02	2.22e-02	1.94e-02	2.46e-02	2.59e-02	2.49e-02	1.75e-02	2.02e-02	1.99e-02
	-$S.D.$	1.52e-01	1.23e-01	1.49e-01	1.39e-01	1.57e-01	1.61e-01	1.58e-01	1.32e-01	1.42e-01	1.41e-01
AlexNet	V^T_{equal}	**167**	**143**	**139**	**169**	**187**	**160**	**160**	**176**	**163**	**150**
	$S\_M^T_{FMI}$	**131.15**	**113.31**	**109.04**	**135.30**	**148.08**	**125.76**	**126.53**	**137.60**	**127.59**	**118.99**
	-Var	7.57e-03	7.53e-03	6.87e-03	8.38e-03	6.02e-03	6.79e-03	6.27e-03	5.23e-03	7.03e-03	6.04e-03
	-$S.D.$	8.70e-02	8.68e-02	8.29e-02	9.15e-02	7.76e-02	8.24e-02	7.92e-02	7.23e-02	8.38e-02	7.77e-02
	$S\_M^T_{JC}$	**109.70**	**95.43**	**91.11**	**115.11**	**124.54**	**105.42**	**106.46**	**115.04**	**106.70**	**100.56**
	-Var	1.59e-02	1.58e-02	1.38e-02	1.75e-02	1.23e-02	1.45e-02	1.28e-02	1.03e-02	1.41e-02	1.28e-02
	-$S.D.$	1.26e-01	1.26e-01	1.17e-01	1.32e-01	1.11e-01	1.20e-01	1.13e-01	1.01e-01	1.19e-01	1.13e-01
	$S\_M^T_{PR}$	**123.10**	**109.06**	**102.31**	**131.53**	**145.66**	**124.69**	**127.87**	**136.72**	**127.40**	**120.04**
	-Var	2.35e-02	2.25e-02	3.02e-02	2.12e-02	1.78e-02	2.04e-02	2.21e-02	1.94e-02	2.23e-02	1.96e-02
	-$S.D.$	1.53e-01	1.50e-01	1.74e-01	1.46e-01	1.33e-01	1.43e-01	1.49e-01	1.39e-01	1.49e-01	1.40e-01
	$S\_M^T_{RR}$	**80.44**	**70.11**	**69.80**	**86.33**	**89.91**	**76.78**	**80.89**	**82.49**	**79.07**	**74.08**
	-Var	2.35e-02	2.56e-02	1.93e-02	2.53e-02	2.12e-02	2.45e-02	2.31e-02	1.88e-02	2.47e-02	2.59e-02
	-$S.D.$	1.53e-01	1.60e-01	1.39e-01	1.59e-01	1.46e-01	1.56e-01	1.52e-01	1.37e-01	1.57e-01	1.61e-01
ResNet-18	V^T_{equal}	**118**	**153**	**151**	**157**	**150**	**161**	**157**	**147**	**172**	**137**
	$S\_M^T_{FMI}$	**92.71**	**121.25**	**119.68**	**125.56**	**115.85**	**126.42**	**122.81**	**115.38**	**136.39**	**109.13**
	-Var	7.28e-03	7.82e-03	7.51e-03	6.78e-03	5.11e-03	5.22e-03	5.18e-03	5.63e-03	5.53e-03	5.92e-03
	-$S.D.$	8.53e-02	8.84e-02	8.66e-02	8.24e-02	7.15e-02	7.23e-02	7.20e-02	7.50e-02	7.43e-02	7.69e-02
	$S\_M^T_{JC}$	**77.48**	**102.14**	**100.77**	**106.47**	**95.82**	**106.07**	**102.51**	**96.42**	**114.98**	**92.45**
	-Var	1.56e-02	1.65e-02	1.55e-02	1.46e-02	1.02e-02	1.12e-02	1.05e-02	1.16e-02	1.14e-02	1.30e-02
	-$S.D.$	1.25e-01	1.29e-01	1.24e-01	1.21e-01	1.01e-01	1.06e-01	1.02e-01	1.08e-01	1.07e-01	1.14e-01
	$S\_M^T_{PR}$	**82.63**	**115.51**	**115.36**	**122.86**	**114.62**	**121.60**	**121.79**	**116.69**	**135.98**	**109.16**
	-Var	3.00e-02	2.99e-02	2.73e-02	2.37e-02	1.67e-02	2.31e-02	1.74e-02	1.70e-02	2.30e-02	2.31e-02
	-$S.D.$	1.73e-01	1.73e-01	1.65e-01	1.54e-01	1.29e-01	1.52e-01	1.32e-01	1.30e-01	1.52e-01	1.52e-01
	$S\_M^T_{RR}$	**59.37**	**79.27**	**76.55**	**78.96**	**68.59**	**77.27**	**73.57**	**68.52**	**84.51**	**67.48**
	-Var	1.89e-02	2.38e-02	1.97e-02	2.71e-02	1.68e-02	2.11e-02	1.95e-02	2.08e-02	1.97e-02	2.37e-02
	-$S.D.$	1.38e-01	1.54e-01	1.40e-01	1.65e-01	1.30e-01	1.45e-01	1.40e-01	1.44e-01	1.40e-01	1.54e-01

determination threshold takes 10%, 20%,..., or 100%, because all the five metrics seem to be stable regardless of how many breakpoints are set. We illustrate this trend (i.e., the row "VGG-16" of Table 7) in Figure 4(a). Besides, we can find that no matter from the perspective of FMI, JC, PR, or RR, the stability of clustering effectiveness (measured by the variance and the standard deviation) does not change significantly as the increase of values of the breakpoint determination threshold. For example, the cell ("VGG-16," "10%," $S\_M^T_{FMI}$-Var) is 7.22e-03, revealing that among 161 "$k == r$" faulty versions obtained by using $SURE^{10\%}_{VGG\text{-}16}$, the variance of the values of FMI is 7.22e-03, and this value changes little as the value of the breakpoint determination threshold increasing. This confirms that it does not matter much if the breakpoint determination threshold takes 10%, 20%,..., or 100%.

Table 8. Performance of SURE on Defects4J

		10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
VGG-16	V_{equal}^{T}	**35**	**37**	**31**	**30**	**31**	**37**	**36**	**28**	**32**	**31**
	$S\_M_{FMI}^{T}$	**34.91**	**36.84**	**30.95**	**29.90**	**30.91**	**36.85**	**35.86**	**27.86**	**31.85**	**30.94**
	-Var	3.62e-05	9.48e-05	2.00e-05	8.89e-05	4.64e-05	8.36e-05	6.82e-05	1.25e-04	9.37e-05	5.43e-05
	-$S.D.$	6.02e-03	9.74e-03	4.47e-03	9.43e-03	6.81e-03	9.14e-03	8.26e-03	1.12e-02	9.68e-03	7.37e-03
	$S\_M_{JC}^{T}$	**34.83**	**36.69**	**30.90**	**29.81**	**30.84**	**36.72**	**35.73**	**27.74**	**31.73**	**30.88**
	-Var	1.45e-04	3.76e-04	7.99e-05	3.10e-04	1.48e-04	2.72e-04	2.47e-04	4.64e-04	3.26e-04	1.85e-04
	-$S.D.$	1.20e-02	1.94e-02	8.94e-03	1.76e-02	1.21e-02	1.65e-02	1.57e-02	2.15e-02	1.80e-02	1.36e-02
	$S\_M_{PR}^{T}$	**33.03**	**34.70**	**30.00**	**28.72**	**29.52**	**35.02**	**33.68**	**25.99**	**29.81**	**30.04**
	-Var	1.51e-02	1.32e-02	7.44e-03	1.25e-02	1.02e-02	1.09e-02	1.73e-02	1.79e-02	1.65e-02	9.46e-03
	-$S.D.$	1.23e-01	1.15e-01	8.62e-02	1.12e-01	1.01e-01	1.04e-01	1.32e-01	1.34e-01	1.28e-01	9.73e-02
	$S\_M_{RR}^{T}$	**33.05**	**34.41**	**29.92**	**28.58**	**29.37**	**34.67**	**33.51**	**25.85**	**29.48**	**29.95**
	-Var	1.48e-02	1.76e-02	8.35e-03	1.57e-02	1.31e-02	1.50e-02	1.87e-02	2.08e-02	2.17e-02	1.12e-02
	-$S.D.$	1.21e-01	1.33e-01	9.14e-02	1.25e-01	1.14e-01	1.22e-01	1.37e-01	1.44e-01	1.47e-01	1.06e-01
AlexNet	V_{equal}^{T}	**41**	**40**	**32**	**34**	**31**	**38**	**34**	**32**	**33**	**32**
	$S\_M_{FMI}^{T}$	**40.81**	**39.90**	**31.92**	**33.96**	**30.87**	**37.80**	**33.90**	**31.97**	**32.91**	**31.89**
	-Var	8.34e-05	2.38e-05	3.13e-05	1.63e-05	7.60e-05	1.09e-04	8.55e-05	8.50e-06	2.59e-05	6.63e-05
	-$S.D.$	9.13e-03	4.87e-03	5.59e-03	4.03e-03	8.72e-03	1.04e-02	9.24e-03	2.91e-03	5.09e-03	8.14e-03
	$S\_M_{JC}^{T}$	**40.63**	**39.80**	**31.83**	**33.92**	**30.75**	**37.63**	**33.82**	**31.94**	**32.84**	**31.79**
	-Var	3.36e-04	1.00e-04	1.37e-04	6.51e-05	2.74e-04	3.97e-04	2.84e-04	3.40e-05	9.77e-05	2.23e-04
	-$S.D.$	1.83e-02	1.00e-02	1.17e-02	8.07e-03	1.65e-02	1.99e-02	1.68e-02	5.83e-03	9.88e-03	1.49e-02
	$S\_M_{PR}^{T}$	**37.64**	**37.66**	**30.35**	**33.25**	**28.83**	**34.83**	**32.77**	**31.25**	**30.91**	**30.21**
	-Var	2.13e-02	1.40e-02	1.32e-02	5.40e-03	1.88e-02	2.00e-02	7.88e-03	5.31e-03	1.52e-02	1.12e-02
	-$S.D.$	1.46e-01	1.18e-01	1.15e-01	7.35e-02	1.37e-01	1.41e-01	8.88e-02	7.29e-02	1.23e-01	1.06e-01
	$S\_M_{RR}^{T}$	**37.44**	**37.58**	**30.35**	**33.17**	**28.58**	**34.58**	**32.61**	**31.25**	**31.01**	**30.10**
	-Var	2.50e-02	1.46e-02	1.32e-02	6.28e-03	2.32e-02	2.49e-02	1.42e-02	5.31e-03	1.48e-02	1.36e-02
	-$S.D.$	1.58e-01	1.21e-01	1.15e-01	7.93e-02	1.52e-01	1.58e-01	1.19e-01	7.29e-02	1.22e-01	1.17e-01
ResNet-18	V_{equal}^{T}	**37**	**31**	**33**	**36**	**34**	**33**	**33**	**29**	**33**	**33**
	$S\_M_{FMI}^{T}$	**36.85**	**30.90**	**32.91**	**35.92**	**33.88**	**32.89**	**32.92**	**28.88**	**32.90**	**32.93**
	-Var	5.65e-05	6.70e-05	2.59e-05	1.73e-05	8.75e-05	4.65e-05	4.26e-05	7.25e-05	6.35e-05	5.31e-05
	-$S.D.$	7.52e-03	8.19e-03	5.09e-03	4.16e-03	9.36e-03	6.82e-03	6.53e-03	8.52e-03	7.97e-03	7.29e-03
	$S\_M_{JC}^{T}$	**36.71**	**30.82**	**32.83**	**35.84**	**33.78**	**32.81**	**32.85**	**28.79**	**32.80**	**32.88**
	-Var	2.22e-04	2.18e-04	1.04e-04	7.47e-05	2.93e-04	1.46e-04	1.64e-04	2.20e-04	2.24e-04	1.63e-04
	-$S.D.$	1.49e-02	1.48e-02	1.02e-02	8.64e-03	1.71e-02	1.21e-02	1.28e-02	1.48e-02	1.50e-02	1.27e-02
	$S\_M_{PR}^{T}$	**34.36**	**29.41**	**31.07**	**34.04**	**32.03**	**30.83**	**31.75**	**27.04**	**31.16**	**32.04**
	-Var	1.53e-02	1.25e-02	1.25e-02	1.18e-02	1.45e-02	1.59e-02	8.42e-03	1.65e-02	1.61e-02	6.37e-03
	-$S.D.$	1.24e-01	1.12e-01	1.12e-01	1.09e-01	1.21e-01	1.26e-01	9.18e-02	1.29e-01	1.27e-01	7.98e-02
	$S\_M_{RR}^{T}$	**34.20**	**29.44**	**30.93**	**34.01**	**31.92**	**30.80**	**31.51**	**26.98**	**31.08**	**31.93**
	-Var	1.71e-02	1.11e-02	1.35e-02	1.17e-02	1.76e-02	1.67e-02	1.32e-02	1.75e-02	1.68e-02	8.46e-03
	-$S.D.$	1.31e-01	1.05e-01	1.16e-01	1.08e-01	1.33e-01	1.29e-01	1.15e-01	1.32e-01	1.30e-01	9.20e-02

A similar trend can be found when AlexNet and ResNet-18 serve as the feature extraction network. Specifically, if we adopt AlexNet or ResNet-18 as the feature extraction network (i.e., the row "AlexNet" or the row "ResNet-18" of Table 7), there is no explicit trend toward change in the five metrics with the increase of x either, as shown in Figure 4(b) and (c), respectively. Besides, the values of the variance and standard deviation are relatively stable as well when AlexNet and ResNet-18 serve as the feature extraction network.

6.1.2 *In Real-World Environments.*

Likewise, let us first use VGG-16 as the feature extraction network, to investigate the impact of the breakpoint determination threshold on the effectiveness of SURE. The results are reported in the row "VGG-16" of Table 8. Similar to the trend observed in simulated environments, we can find that the effectiveness of SURE is likely to be stable, regardless of how many breakpoints are set according to the suspiciousness of program statements. For

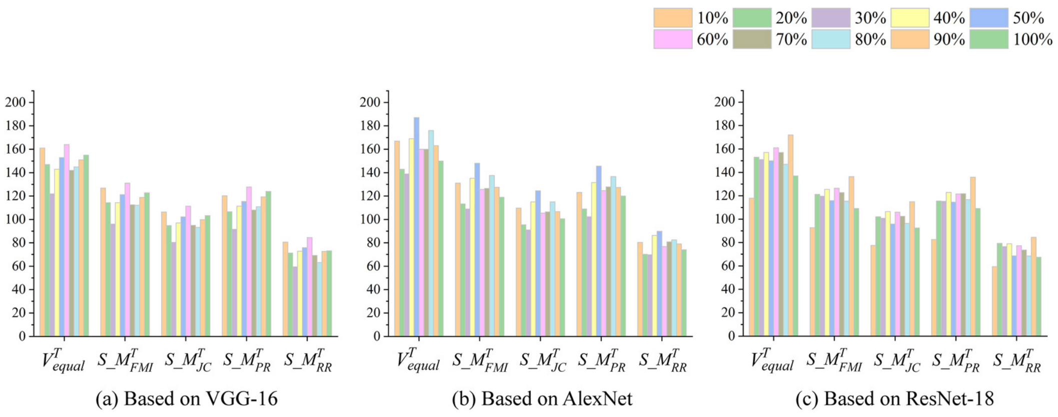


Fig. 4. The impact of the breakpoint determination threshold on SURE on SIR.

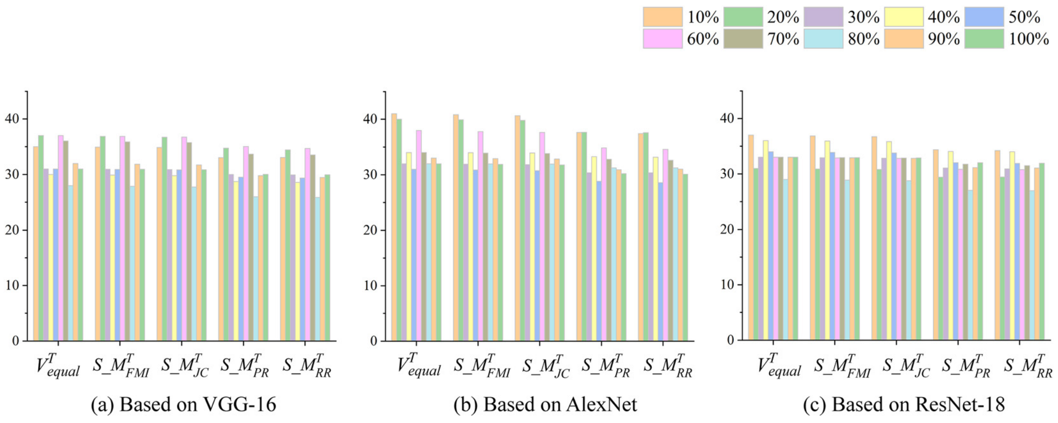


Fig. 5. The impact of the breakpoint determination threshold on SURE on Defects4J.

example, when the threshold is set from 10% to 100% with 10% increments, the value of V^T_{equal} is 35, 37, 31, 30, 31, 37, 36, 28, 32, and 31. And the values of $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$ seem to be not largely influenced by the breakpoint determination threshold either. We depict this trend in Figure 5(a). When the feature extraction model is replaced with AlexNet and ResNet-18, this conclusion can still be maintained, as shown in the row "AlexNet" and the row "ResNet-18" of Table 8, as well as Figure 5(b) and (c), respectively. Besides, the values of the variance and standard deviation do not change significantly as the increase of values of the breakpoint determination threshold, no matter from the perspective of FMI, JC, PR, or RR.

The result of RQ1 can be explained to mean that the memory information collected at the Top-10% riskiest statements is already sufficient to represent failures. This result confirms our intuition of the breakpoint determination phase (Section 4.1), i.e., statements with higher risk values are more likely to be faulty, and runtime memory information gathered at these positions could have a stronger capability to reveal faults, and thus can contribute more to representing failures. Moreover, the result of RQ1 also reveals the necessity of the breakpoint determination component of SURE. Specifically, more memory information cannot contribute more to failure representation, thus we can first use SBFL to determine a few breakpoints, and then collect memory information within a

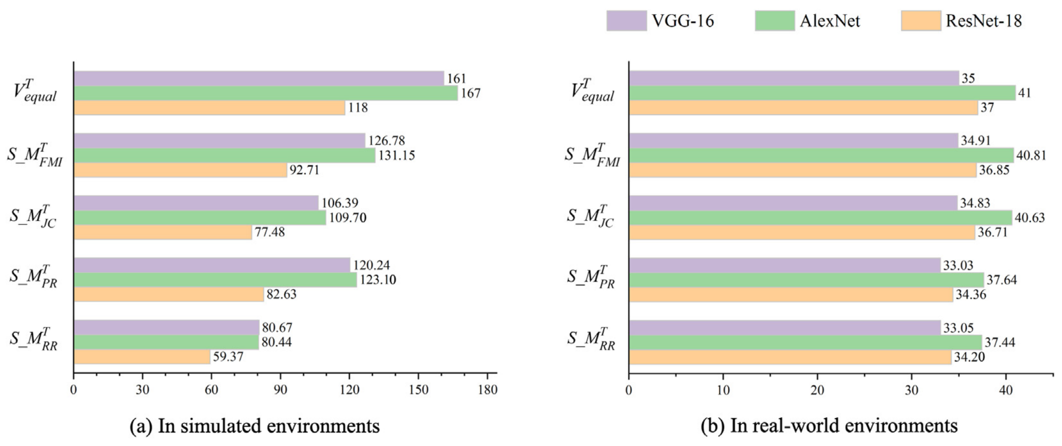


Fig. 6. The impact of the feature extraction network on SURE.

small range. This is practically important because the overhead of running SBFL is less than that of collecting runtime memory information. To put it another way, we preliminarily use a lower-cost technique to avoid the following procedure with relatively higher costs, while such a procedure could not bring gains for our approach. Therefore, *in the following RQs, the investigation will be based on the breakpoint determination threshold of Top-10%.*

6.2 RQ2: The Adopted Feature Extraction Network

In addition to the breakpoint determination threshold, another configurable hyperparameter of SURE is the feature extraction network. In this RQ, we select three deep CNNs, VGG-16, AlexNet, and ResNet-18, to investigate how the effectiveness of SURE will be influenced when adopting different networks.

6.2.1 In Simulated Environments. We first analyze the results in simulated environments, as shown in the column "10%" of Table 7. We can find that there are 167 "$k == r$" faulty versions when using AlexNet as the feature extraction network, while 161 and 118 ones when using VGG-16 and ResNet-18, respectively. As for the metrics $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$, AlexNet also dominates VGG-16 and ResNet-18 (with only one exception where $S\_M_{RR}^{SURE^{10\%}_{VGG\text{-}16}}$: 80.67 is slightly higher than $S\_M_{RR}^{SURE^{10\%}_{AlexNet}}$: 80.44). We illustrate these results in Figure 6(a) as well for clear understanding. We can find that in simulated environments, AlexNet can better extract features for PMS images thus deliver more promising failure indexing outcomes.

6.2.2 In Real-World Environments. Now we navigate to real-world environments, the results are given in the column "10%" of Table 8. We can find that there are 41 "$k == r$" faulty versions when using AlexNet as the feature extraction network, while 35 and 37 ones when using VGG-16 and ResNet-18, respectively. As for the metrics $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$, AlexNet also dominates VGG-16 and ResNet-18. We illustrate these results in Figure 6(b) as well for clear understanding. Similar to the conclusion in simulated environments, we can find that AlexNet is more suitable for extracting features for PMS images.

A possible explanation for the promise of AlexNet in our experiments could be its relatively shallow architecture. Concretely speaking, PMS images are typically in a simple form (not too many complex features involved in PMS images), it can be naturally conjectured that a shallow CNN might be enough to extract their features. Among the selected three networks, AlexNet consists of eight layers including five convolutional ones, while VGG-16 and ResNet-18 both have much deeper architectures than AlexNet. Thus, it is intuitive that SURE could perform better when configuring AlexNet as the feature extraction network. Therefore, *in the following RQs, the investigation will be based on the feature extraction network of AlexNet.*

6.3 RQ3: The Competitiveness of SURE

In this RQ, we investigate to which extent SURE exceeds the existing most prevalent and advanced failure indexing techniques. Given the conclusions of RQ1 and RQ2, we configure 10% as the breakpoint determination threshold and AlexNet as the feature extraction network for SURE in this RQ. To put it another way, we will use $SURE^{10\%}_{AlexNet}$ to compare with baseline techniques (for convenience, we use the term "$SURE$" to denote "$SURE^{10\%}_{AlexNet}$" hereafter).

6.3.1 Clustering Effectiveness in Simulated Environments. The results in simulated environments from a global perspective are given in the row "Total" of Table 9 and Figure 7(a). It can be observed that $SURE$ outperforms all the baseline techniques. Specifically, regarding the capability of faults number estimation, $SURE$ can correctly predict the number of faults on 167 faulty versions on SIR, with 518.52%, 421.88%, 421.88%, 209.26%, 101.20%, and 25.56% improvements compared with $Cov_{hit\text{-}Cos}$ (27), $Cov_{hit\text{-}Jacc}$ (32), $Cov_{hit\text{-}Eucli}$ (32), Cov_{count} (54), $MSeer$ (83), and $MSeer_{EFF\text{10-83}}$ (133), respectively. $SURE$ consistently exceeds the baselines on all four clustering metrics. For instance, if we focus on the comparison between $SURE$ and $Cov_{hit\text{-}Eucli}$, improvements are 418.17%, 414.06%, 431.06%, and 439.14%, regarding $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$, respectively. Considering the four clustering metrics globally, the average improvement of $SURE$ over $Cov_{hit\text{-}Eucli}$ is 425.61%. Similarly, in the contexts of comparing $SURE$ with $Cov_{hit\text{-}Cos}$, $Cov_{hit\text{-}Jacc}$, Cov_{count}, $MSeer$, and $MSeer_{EFF\text{10-83}}$, the average improvements can be calculated as 549.28%, 445.55%, 187.18%, 105.20%, and 19.27%, respectively. Besides, as mentioned in Section 5.3, all SIR faulty versions can be divided into 1-bug, 2-bug, 3-bug, 4-bug, and 5-bug ones. Here we also give the experimental results of these five sub-environments, as shown in the rows "1-bug" ∼ "5-bug" of Table 9, respectively. It can be observed that in each sub-part, SURE can outperform baseline techniques as well.

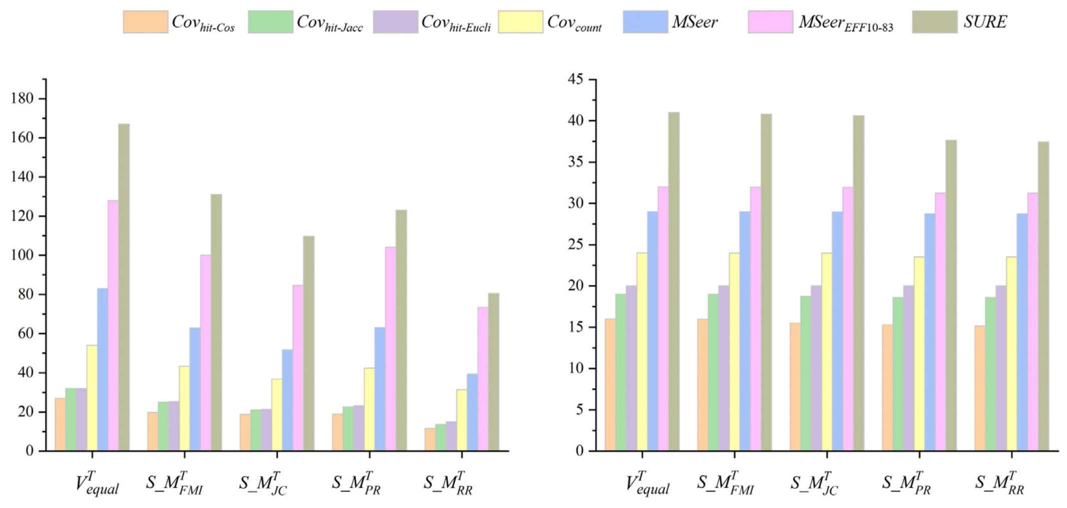
6.3.2 Clustering Effectiveness in Real-World Environments. The results in real-world environments are given in Table 10 and Figure 7(b). It can be observed that $SURE$ outperforms all the baseline techniques. Specifically, regarding the capability of faults number estimation, $SURE$ can correctly predict the number of faults on 41 faulty versions on Defects4J, with 156.25%, 115.79%, 105.00%, 70.83%, 41.38%, and 24.24% improvements compared with $Cov_{hit\text{-}Cos}$ (16), $Cov_{hit\text{-}Jacc}$ (19), $Cov_{hit\text{-}Eucli}$ (20), Cov_{count} (24), $MSeer$ (29), and $MSeer_{EFF\text{10-83}}$ (33), respectively. $SURE$ consistently exceeds the baselines on all four clustering metrics. For instance, if we focus on the comparison between $SURE$ and $Cov_{hit\text{-}Eucli}$, improvements are 104.05%, 103.15%, 88.20%, and 87.20%, regarding $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$, respectively. Considering the four clustering metrics globally, the average improvement of $SURE$ over $Cov_{hit\text{-}Eucli}$ is 95.65%. Similarly, in the contexts of comparing $SURE$ with $Cov_{hit\text{-}Cos}$, $Cov_{hit\text{-}Jacc}$, Cov_{Count}, $MSeer$, and $MSeer_{EFF\text{10-83}}$, the average improvements can be calculated as 152.70%, 108.79%, 64.81%, 35.53%, and 20.21%, respectively. Notice that the faults provided in Defects4J are from the real world, the way of combining multiple single-faults in Defects4J differs from that in SIR. Therefore, the numbers of generated 1-bug, 2-bug, 3-bug, 4-bug, and 5-bug Defects4J faulty versions are relatively small, thus the experimental results of Defects4J are presented as a whole.

6.3.3 AP. The evaluation results of the AP are given in Table 11, in which the baseline value is the proportion of positive samples. It can be observed that in SIR, SURE obtains an AP value of 0.79,

Table 9. Clustering Effectiveness of SURE in Simulated Environments

# Faults	T	V^T_{equal}	$S\_M^T_{FMI}$	$S\_M^T_{JC}$	$S\_M^T_{PR}$	$S\_M^T_{RR}$
1-bug	$Cov_{hit\text{-}Cos}$	0	0.00	0.00	0.00	0.00
	$Cov_{hit\text{-}Jacc}$	0	0.00	0.00	0.00	0.00
	$Cov_{hit\text{-}Eucli}$	1	1.00	1.00	1.00	1.00
	Cov_{count}	3	3.00	3.00	3.00	3.00
	$MSeer$	4	4.00	4.00	4.00	4.00
	$MSeer_{EFF\text{10-83}}$	7	7.00	7.00	7.00	7.00
	SURE	**9**	**9.00**	**9.00**	**9.00**	**9.00**
2-bug	$Cov_{hit\text{-}Cos}$	11	8.33	8.01	8.07	4.45
	$Cov_{hit\text{-}Jacc}$	13	9.84	8.32	8.74	5.38
	$Cov_{hit\text{-}Eucli}$	13	9.92	8.38	9.21	5.53
	Cov_{count}	20	16.23	13.77	15.98	10.87
	$MSeer$	29	23.02	16.86	23.43	12.86
	$MSeer_{EFF\text{10-83}}$	41	31.97	24.38	33.76	22.01
	SURE	**51**	**43.26**	**32.08**	**38.99**	**22.35**
3-bug	$Cov_{hit\text{-}Cos}$	10	7.46	7.17	7.17	4.36
	$Cov_{hit\text{-}Jacc}$	11	8.76	7.16	7.83	4.57
	$Cov_{hit\text{-}Eucli}$	9	7.07	6.12	6.54	3.69
	Cov_{count}	14	11.09	8.79	10.89	8.13
	$MSeer$	23	16.16	13.05	16.37	10.17
	$MSeer_{EFF\text{10-83}}$	34	24.62	20.36	26.57	19.61
	SURE	**42**	**30.15**	**25.33**	**30.55**	**19.79**
4-bug	$Cov_{hit\text{-}Cos}$	4	2.66	2.29	2.32	1.75
	$Cov_{hit\text{-}Jacc}$	4	3.11	2.67	2.86	1.92
	$Cov_{hit\text{-}Eucli}$	5	3.92	3.27	3.53	2.42
	Cov_{count}	9	7.52	6.03	7.43	5.07
	$MSeer$	15	11.14	9.97	11.21	7.02
	$MSeer_{EFF\text{10-83}}$	24	19.85	16.03	19.94	13.94
	SURE	**30**	**24.18**	**21.24**	**24.09**	**16.31**
5-bug	$Cov_{hit\text{-}Cos}$	2	1.31	1.30	1.34	0.97
	$Cov_{hit\text{-}Jacc}$	4	3.26	2.87	3.09	1.80
	$Cov_{hit\text{-}Eucli}$	4	3.40	2.57	2.90	2.28
	Cov_{count}	8	5.49	5.16	5.07	4.23
	$MSeer$	12	8.54	7.84	8.07	5.21
	$MSeer_{EFF\text{10-83}}$	27	20.94	19.21	18.25	11.48
	SURE	**35**	**24.56**	**22.05**	**20.47**	**12.99**
Total	$Cov_{hit\text{-}Cos}$	27	19.76	18.77	18.90	11.53
	$Cov_{hit\text{-}Jacc}$	32	24.97	21.02	22.52	13.67
	$Cov_{hit\text{-}Eucli}$	32	25.31	21.34	23.18	14.92
	Cov_{count}	54	43.33	36.75	42.37	31.30
	$MSeer$	83	62.86	51.72	63.08	39.26
	$MSeer_{EFF\text{10-83}}$	133	104.38	86.98	105.52	74.04
	SURE	**167**	**131.15**	**109.70**	**123.10**	**80.44**

higher than both the baseline value (0.27) and the other compared techniques (from 0.26 to 0.63). Similarly, in Defects4J, SURE obtains an AP value of 0.85, consistently exceeds the baseline value (0.22) and outperforms the other compared techniques (from 0.27 to 0.68). These results reveal that



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

Fig. 7. Clustering effectiveness of SURE.

Table 10. Clustering Effectiveness of SURE in Real-World Environments

T	V^T_{equal}	$S\_M^T_{FMI}$	$S\_M^T_{JC}$	$S\_M^T_{PR}$	$S\_M^T_{RR}$
$Cov_{hit\text{-}Cos}$	16	15.97	15.50	15.30	15.15
$Cov_{hit\text{-}Jacc}$	19	19.00	18.75	18.61	18.59
$Cov_{hit\text{-}Eucli}$	20	20.00	20.00	20.00	20.00
Cov_{count}	24	23.98	23.96	23.50	23.50
MSeer	29	28.99	28.98	28.75	28.75
$MSeer_{EFF10\text{-}83}$	33	32.95	32.86	32.25	32.10
SURE	**41**	**40.81**	**40.63**	**37.64**	**37.44**

Table 11. AP of SURE

	SIR		**Defects4J**	
	AP	Baseline	AP	Baseline
$Cov_{hit\text{-}Cos}$	0.26	0.27	0.27	0.22
$Cov_{hit\text{-}Jacc}$	0.26	0.27	0.32	0.22
$Cov_{hit\text{-}Eucli}$	0.29	0.27	0.32	0.22
Cov_{count}	0.34	0.27	0.43	0.22
MSeer	0.55	0.27	0.65	0.22
$MSeer_{EFF10\text{-}83}$	0.63	0.27	0.68	0.22
SURE	**0.79**	0.27	**0.85**	0.22

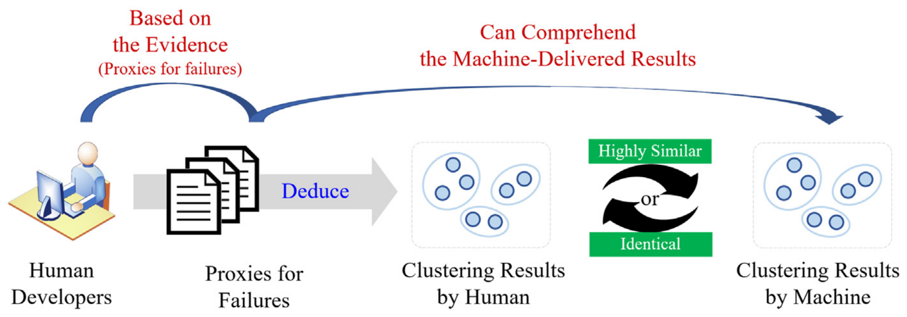


Fig. 8. Quantitatively measure the comprehensibility of failure indexing techniques.

even though the evaluation is extended to be independent of clustering algorithms, SURE can still present promising performance.

As a reminder, $MSeer_{EFF10\text{-}83}$ is not an existing published failure indexing technique, it is manually created by us to further evaluate the competitiveness of SURE. *MSeer* is the state-of-the-art technique in the field of failure indexing.

6.4 RQ4: The Comprehensibility of SURE

In Section 1, we point out that one of the two longstanding challenges in failure indexing is the lack of comprehensibility, that is, the evidence (i.e., the proxies for failed test cases) for how failure indexing results were obtained is hard to comprehend for human developers, which could prevent the results from being applied. The comprehensibility of a failure indexing technique essentially lies in the comprehensibility of the failure proximity, because it is the failure proximity that represents failures and hence provides human developers with the evidence of failure indexing. Here we describe the comprehensibility of failure indexing techniques as follows:

A failure indexing technique is considered to have strong comprehensibility, if given the failure indexing result as well as the evidence (i.e., the proxies for failed test cases) for how this result was obtained, ***developers can be immediately convinced by the result based on the evidence.***

But this description cannot directly guide the investigation of RQ4, because it is subjective for developers to determine whether they are convinced by the result or not, and it is difficult to quantitatively measure the extent to which they are convinced. To tackle this problem, we propose the following strategy to quantify the comprehensibility of a failure indexing technique:

Providing developers with only the evidence (i.e., the proxies for failed test cases) of a failure indexing process, and letting them manually cluster failed test cases according to the evidence. ***The closer their manual clustering outcomes are to the failure indexing results, the better they comprehend the evidence****, and thus, this failure indexing technique has stronger comprehensibility.*

The intuition behind this strategy can be described in Figure 8. Specifically, if only the evidence of a failure indexing process is provided to developers, and the result of their manual clustering based on that evidence is similar or even identical to the result of the failure indexing technique, then when the failure indexing result as well as the evidence are provided together, they are able to comprehend the evidence.

Table 12. Nine Failure Indexing Tasks

Project	**Language**	**# Failures**	**# Faults**	**Failure representation for developers**		
				PMS	Ranking list[a]	Coverage[b]
Task-1	C	10	1	***1***, *4, 7, 10, 13, 16*	*2, 5, 8, 11, 14, 17*	*3, 6, 9, 12, 15, 18*
Task-2	C	29	2	**1**, 4, 7, 10, 13, 16	2, 5, 8, 11, 14, 17	3, 6, 9, 12, 15, 18
Task-3	C	10	3	**1**, 4, 7, 10, 13, 16	2, 5, 8, 11, 14, 17	3, 6, 9, 12, 15, 18
Task-4	C	19	4	3, 6, 9, 12, 15, 18	**1**, 4, 7, 10, 13, 16	2, 5, 8, 11, 14, 17
Task-5	C	17	5	3, 6, 9, 12, 15, 18	**1**, 4, 7, 10, 13, 16	2, 5, 8, 11, 14, 17
Task-6	Java	6	2	3, 6, 9, 12, 15, 18	**1**, 4, 7, 10, 13, 16	2, 5, 8, 11, 14, 17
Task-7	Java	3	3	2, 5, 8, 11, 14, 17	3, 6, 9, 12, 15, 18	**1**, 4, 7, 10, 13, 16
Task-8	Java	4	4	2, 5, 8, 11, 14, 17	3, 6, 9, 12, 15, 18	**1**, 4, 7, 10, 13, 16
Task-9	Java	5	5	2, 5, 8, 11, 14, 17	3, 6, 9, 12, 15, 18	**1**, 4, 7, 10, 13, 16

[a]"Ranking list" denotes "Suspiciousness ranking lists." The same in Table 13.
[b]"Coverage" denotes "Coverage vectors of execution frequency." The same in Table 13.

As such, the clustering effectiveness made by human developers can serve as the comprehensibility of a failure indexing technique. Hence, the evaluation metrics introduced in Section 5.4, i.e., V^T_{equal}, $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, and $S\_M^T_{RR}$, can be directly employed in this RQ.

We recruit 18 participants to perform the human study, 15 of whom are graduate students in Computer Science from Wuhan University and have at least four years of programming experience (Participant-1 ∼ Participant-15). The remaining three are from top-tier Internet companies in China and have five ∼ eight years of programming experience (Participant-16 ∼ Participant-18). These 18 participants are selected to ensure they all have expertise in both C and Java, and they are not compensated for their participation. They are requested to manually finish nine failure indexing tasks (each task corresponds to an r-bug faulty program, where r = 1, 2, 3, 4, or 5). Specifically, in each task, only a series of failed test cases are provided to participants, we did not tell participants how many fault(s) triggered these failures and the mutual relationship among these failures. Participants will manually divide all failures through two steps: (1) Estimating these failures should be divided into how many clusters, i.e., there are how many underlying faults behind these failures. (2) Manually dividing failures according to the estimated number of clusters.

As for each task, its failed test cases are given in one of three forms of representation: PMS images, suspiciousness ranking lists, and coverage vectors of execution frequency. Specifically, PMS images are the new form of failure representation proposed in this article, suspiciousness ranking lists are the failure representation of SD-based proximities (the most advanced strategy), and coverage vectors are the failure representation of CC-based failure proximities (the most prevalent strategy). Notice that at the beginning of the human study, we briefly introduce failure representation forms of ranking lists and coverage to participants, showing them how a failure is translated into a ranking list or a coverage vector (taking about five minutes and one minute, respectively). To put it another way, participants become aware of the meaning of failure representation forms of ranking lists and coverage in advance. On the contrary, we do not introduce how a failure is translated into a PMS image to participants, this is because we want to know how well human developers can comprehend failures in the form of PMS images without prior knowledge, to check the human-friendliness of PMS images.

We concisely introduce the nine tasks in Table 12, including the programming language, the number of observed failures in testing, as well as the number of underlying faults. *From the perspective of tasks*, each task will be carried out by all of the 18 participants, among them, six are based on PMS images, six are based on suspiciousness ranking lists, and six are based on coverage vectors. For example, we can find that for Task-1, Participants 1, 4, 7, 10, 13, 16 will be based on PMS

Table 13. Comprehensibility of Three Forms of Failure Representation

		Participant																		Total
		No. 1	No. 2	No. 3	No. 4	**No. 5**	No. 6	No. 7	No. 8	No. 9	No. 10	No. 11	No. 12	No. 13	No. 14	No. 15	No. 16	No. 17	No. 18	
PMS	V^T_{equal}	2	3	2	3	**3**	3	3	3	3	2	3	2	1	3	3	3	3	3	48
	$S\_M^T_{FMI}$	2.00	3.00	2.00	3.00	**3.00**	3.00	3.00	3.00	3.00	2.00	3.00	2.00	1.00	3.00	3.00	3.00	3.00	3.00	48.00
	$S\_M^T_{JC}$	2.00	3.00	2.00	3.00	**3.00**	3.00	3.00	3.00	3.00	2.00	3.00	2.00	1.00	3.00	3.00	3.00	3.00	3.00	48.00
	$S\_M^T_{PR}$	2.00	3.00	2.00	3.00	**3.00**	3.00	3.00	3.00	3.00	2.00	3.00	1.96	1.00	3.00	3.00	3.00	3.00	3.00	47.96
	$S\_M^T_{RR}$	2.00	3.00	2.00	3.00	**3.00**	3.00	3.00	3.00	3.00	2.00	3.00	1.93	1.00	3.00	3.00	3.00	3.00	3.00	47.93
	Average Time (s)	5.41	3.08	1.50	6.22	**1.17**	3.24	1.39	3.42	4.43	5.63	3.11	2.62	3.02	2.33	2.33	4.06	1.83	2.83	3.20
Ranking list	V^T_{equal}	2	0	3	2	**0**	1	0	1	2	2	1	1	1	0	3	0	0	1	20
	$S\_M^T_{FMI}$	1.99	0.00	3.00	1.99	**0.00**	1.00	0.00	0.99	2.00	1.99	0.99	1.00	1.00	0.00	3.00	0.00	0.00	1.00	19.95
	$S\_M^T_{JC}$	1.99	0.00	3.00	1.99	**0.00**	1.00	0.00	0.98	2.00	1.99	0.98	1.00	1.00	0.00	3.00	0.00	0.00	1.00	19.93
	$S\_M^T_{PR}$	1.81	0.00	3.00	1.62	**0.00**	1.00	0.00	0.46	2.00	1.73	0.55	1.00	1.00	0.00	3.00	0.00	0.00	1.00	18.17
	$S\_M^T_{RR}$	1.81	0.00	3.00	1.66	**0.00**	1.00	0.00	0.40	2.00	1.71	0.46	1.00	1.00	0.00	3.00	0.00	0.00	1.00	18.04
	Average Time (s)	14.33	25.37	18.50	23.21	**34.92**	21.08	27.38	44.98	13.92	13.11	20.63	16.83	4.69	33.35	6.33	23.69	31.45	21.42	21.96
Coverage	V^T_{equal}	0	2	1	1	**1**	0	2	2	0	2	2	1	3	2	0	1	1	0	21
	$S\_M^T_{FMI}$	0.00	1.99	1.00	1.00	**1.00**	0.00	2.00	1.99	0.00	2.00	1.99	0.99	3.00	2.00	0.00	1.00	1.00	0.00	20.96
	$S\_M^T_{JC}$	0.00	1.99	1.00	1.00	**1.00**	0.00	2.00	1.99	0.00	2.00	1.98	0.99	3.00	2.00	0.00	1.00	1.00	0.00	20.95
	$S\_M^T_{PR}$	0.00	1.72	1.00	1.00	**1.00**	0.00	2.00	1.72	0.00	2.00	1.62	0.48	3.00	1.87	0.00	1.00	1.00	0.00	19.41
	$S\_M^T_{RR}$	0.00	1.66	1.00	1.00	**1.00**	0.00	2.00	1.72	0.00	2.00	1.66	0.47	3.00	1.87	0.00	1.00	1.00	0.00	19.38
	Average Time (s)	12.00	16.86	38.59	20.17	**14.26**	46.63	20.33	35.40	34.31	24.72	14.78	21.90	3.33	12.93	12.71	15.75	16.33	41.33	22.35

to perform failure indexing, Participants 2, 5, 8, 11, 14, 17 will be based on suspiciousness ranking lists, and Participants 3, 6, 9, 12, 15, 18 will be based on coverage vectors, as shown by the italic numbers in Table 12. Additionally, *from the perspective of participants*, each developer will handle all of the nine tasks, among them, three for each of the three forms of representation. For example, we can find that Participant-1 handles Task-1, Task-2, and Task-3 based on PMS images, handles Task-4, Task-5, and Task-6 based on suspiciousness ranking lists, and handles Task-7, Task-8, and Task-9 based on coverage vectors, as shown by the bold numbers in Table 12. As such, each form of representation will be carried out for 54 times, as shown in the columns "PMS," "Ranking list," and "Coverage" in Table 12.

The performance of human developers is given in Table 13.[12] Take Participant-5 as an example, as shown in the bold column "Participant-5" of Table 13. Participant-5 carries out nine failure indexing tasks with three PMS-based, three ranking list-based, and three coverage-based ones. As for the three PMS-based faulty versions, Participant-5 correctly estimates the number of faults on all of them, i.e., the value of the cell ("PMS," "V^T_{equal}," "Participant-5") is three. The values of the four clustering metrics, i.e., $S\_M^T_{FMI}$, $S\_M^T_{JC}$, $S\_M^T_{PR}$, $S\_M^T_{RR}$, are both 3.00. Considering that the maximum values of FMI, JC, PR, and RR are 1.00, this result indicates that Participant-5 can achieve perfect failure indexing based on PMS. But if based on the other forms of failure representation, Participant-5 performs much worse. Specifically, "$k == r$" cannot be obtained on any faulty version based on the representation of ranking lists, and only one faulty version can be properly handled based on the representation of coverage. Moreover, the time cost of manual failure indexing is also different among the three forms of representation. For example, Participant-5 takes 1.17s on average to index each failure when it is represented as a PMS image, while takes 34.92s and 14.26s on average when a failure is represented as a ranking list and a coverage vector, respectively. That is to say, the proposed failure representation of PMS images enables developers to identify failures with the same/different root cause(s) at a glance, further demonstrating its intuition and friendliness toward human developers.

The other participants perform similarly to Participant-5: they all tend to achieve better failure indexing with a lower time cost based on the failure representation of PMS, compared with using the other two existing forms of representation. We summarize the human performance based on

[12] For a faulty version, consistent with the strategy in Section 5.4, we analyze the clustering effectiveness only when "$k == r$", namely, the number of faults is correctly predicted.

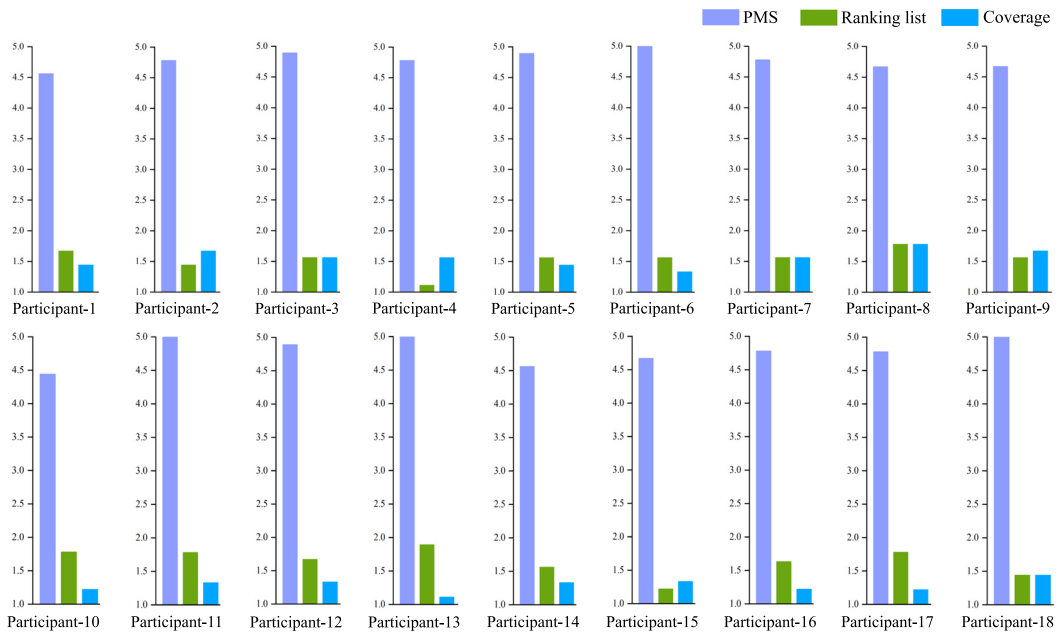
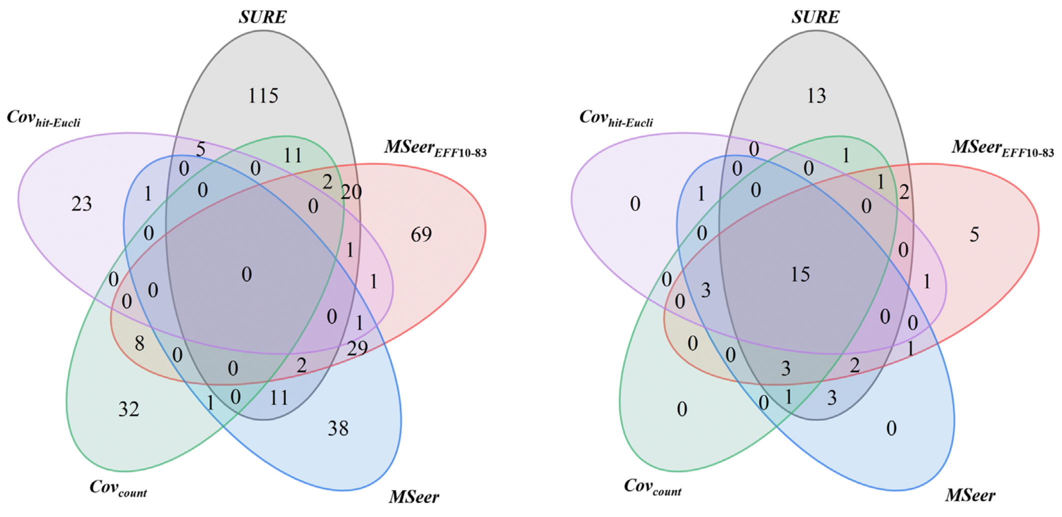


Fig. 9. The utility of failure indexing outcomes.

three forms of failure presentation in the column "Total" of Table 13. Specifically, regarding the faults number estimation, human developers perform 140.00% and 128.57% better when using PMS (48 out of 54 times) than when using ranking lists (20 out of 54 times) and coverage (21 out of 54 times), respectively. And the four clustering metrics show similar magnitude of improvements. The time costs are 85.43% and 85.68% cheaper when using PMS (3.20s) than when using ranking lists (21.96s) and coverage (22.35s), respectively. Revisiting the statements at the beginning of this sub-section, we can conclude that the proposed failure representation of PMS has stronger comprehensibility compared with the two most advanced and prevalent techniques.

Apart from the comprehensibility of PMS images, we are also concerned about the developers' perception of the utility of the outcome of SURE, that is, the extent to which they will utilize failure indexing outcomes (i.e, resulting clusters) when failures are in the form of PMS images. To this end, we design an extra questionnaire that requires participants to give a score for the outcomes of the mentioned failure indexing tasks, to quantitatively reflect the possibility of adopting the outcome. Inspired by the design of human studies of prior works in the field of software testing and debugging [77, 104], the extra questionnaire is based on a five-point Likert scale [41], where one means strongly disagree to utilize and five means strongly agree to utilize. Similar with the aforementioned human study, we also employ the failure representation forms of ranking lists and coverage as baselines. The results are given in Figure 9. It can be seen that when failures are represented as PMS images, the scores of utilities are all between four and five (the minimum value is 4.44 given by Participant-10, and the maximum value is 5.00 given by Participant-6, Participant-11, Participant-13, and Participant-18). On the contrary, if failures are represented as ranking lists or coverage, the scores of utilities are all less than two. These results show a consensus among all participants: given failure indexing outcomes with the failure representation form of PMS images, they are more likely to utilize the outcome compared with failure representation forms of ranking lists and coverage.



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

Fig. 10. The divergence of the "$k == r$" faulty versions.

At the end of the human study, we interview the participants. The interview questions are twofold: (1) How do you think about the benefit of PMS images in comprehending failure indexing outcomes? And (2) Why are you more willing to utilize failure indexing outcomes when failures are represented as PMS images? The interview is carried out in person or by telephone, with the first three authors asking questions. We summarize the feedback they give and find that almost all interviewees mention two points: (1) Based on PMS images, they can immediately justify how many clusters the provided failures should be categorized into, and how they should be divided, because the characteristic of visualization of PMS images is highly in tune with human intuition. But when the other two representations are used, it always takes a lot of time to recognize failures, and the process is very tough. (2) If failures are represented as PMS images, they can be better convinced by failure indexing results and thus further apply them to the following debugging tasks. This is because PMS images provide them with a more intuitive way to become aware of why a pair of failures should (or should not) be put together at a glance, and this boosts their confidence in failure indexing results.

7 Discussion

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

7.1 Faulty Versions Uniquely Handled by SURE

In the comparison between SURE and baseline techniques, we find that some faulty versions could be only handled by a specific technique. That is, those "$k == r$" faulty versions achieved by different techniques are not exactly the same. It is intuitive that the more faulty versions a failure indexing technique is able to handle solely, the better this technique is. We further inspect SURE and the baseline techniques from this heuristic aspect, the results are given in Figure 10 (Among the three variants of Cov_{hit}, we select $Cov_{hit\text{-}Eucli}$ as the representative due to its higher prevalence than the other two).

In Figure 10(a), we can find that of all SIR faulty versions in the test, 115 can be handled by SURE only, while 69, 38, 32, and 23 can be handled by $MSeer_{EFF\text{10-83}}$, $MSeer$, Cov_{count}, and $Cov_{hit\text{-}Eucli}$ only, respectively. A similar observation can be drawn from Figure 10(b): of all Defects4J faulty versions, 13 can be handled by SURE only, while five, zero, zero, and zero can be handled by $MSeer_{EFF\text{10-83}}$, $MSeer$, Cov_{count}, and $Cov_{hit\text{-}Eucli}$ only, respectively. The results reveal that no failure indexing technique can be fully dominated by others, indicating a potential future direction of combining the advantages of different classes of failure proximities together. Despite the respective superiority of various techniques, SURE can solely handle more faulty versions than the other techniques. In particular, it exceeds 66.67% and 160.00% by $MSeer_{EFF\text{10-83}}$, the best-performed baseline technique, in simulated and real-world environments, respectively.

7.2 Overhead of SURE

The time costs of SURE mainly involve three phases, namely, the failure representation, the distance measurement (including the model training and the deployment), and the clustering. According to our analyses, as for the failure representation, SURE takes 3.99 minutes and 5.90 minutes on average to collect runtime memory information during the execution of a failed test case, on SIR and Defects4J faulty versions, respectively. Then, SURE takes 0.18s and 0.27s on average to convert a set of memory information to a PMS image, on SIR and Defects4J faulty versions, respectively. As for the distance measurement, SURE takes 23 minutes and 35 seconds to train the Siamese-based model.[13] Once the model is trained, 0.01s and 0.06s on average are taken to predict the distance between a pair of PMS images, on SIR and Defects4J faulty versions, respectively. After these two steps are ready, the clustering process typically takes only a few seconds. To summarize, the overhead of SURE mainly lies in querying runtime memory information at preset breakpoints.

As pioneers have pointed out, failure indexing is essential yet very costly when it comes to manual debugging in the real world. "*Experienced developers can manually examine every failure and determine the culprit fault, but this is apparently too expensive*", pioneers concluded [58]. In contrast, SURE can complete this task automatically and hence can save a great deal of manual effort, which is far less expensive than manual jobs. In particular, SURE represents a failed test case as a PMS image, which can be smoothly comprehended by human developers at a glance and thus can boost the comprehensibility of failure indexing outcomes. This can be of great importance because if developers are not convinced by failure indexing outcomes, they will take much time to verify their correctness. Even though coverage vectors (the failure representation of the CC-based strategy) and statement ranking lists (the failure representation of the SD-based strategy) can also provide insights for developers to comprehend failure indexing outcomes to an extent, they can be human labor-intensive. To put it another way, we transfer the costs that would have been borne by humans to machines. It is true that as compared with CC and SD-based techniques, SURE needs higher costs. However, in return, SURE delivers better performance and convenience. Therefore, the cost of SURE is acceptable: more sophisticated fingerprinting is naturally accompanied by higher overhead, i.e., "*no free lunch*" [58].

8 Threats to Validity

Our experiments are subject to several threats to validity.

Threats to internal validity relate to the faulty versions we focus on to determine the clustering effectiveness of a failure indexing technique. In the evaluation of clustering effectiveness, we primarily consider those faulty versions that satisfy the condition of "$k == r$," i.e., where the

[13]The trained model is available in our repository, it can be directly applied in future failure indexing jobs.

predicted number of faults k is equal to the real number of faults r. Actually, "$k == r$" is ideal but not necessary for parallel debugging to work in practice. This is because if k exceeds r, k developers will be employed to locate r faults, and parallel debugging can work at the cost of human labor (k - r developers are redundant). On the contrary, if k is less than r, parallel debugging can also work at the cost of more than one iteration of debugging. Filtering these faulty versions in the evaluation of clustering effectiveness is because comparing k generated clusters with r oracle clusters when "k != r" is difficult, and moreover, prior studies have pointed out that the performance of a failure indexing technique can be mainly determined by those "$k == r$" faulty versions [27, 73], that is, the contribution of "k != r" ones is marginal. Nonetheless, apart from evaluating failure indexing techniques from the perspective of clustering effectiveness, we also utilize another metric, AP, to deeply assess the effectiveness of PMS images by measuring the extent to which the failure representation of PMS images prioritizes higher similarity for relevant failure pairs over irrelevant ones. Such evaluation is independent of clustering algorithms and thus can include both "$k == r$" and "k != r" faulty versions. Therefore, this threat to validity is acceptable.

Threats to external validity relate to the generalization capability of SURE. Specifically, given that the evaluation of the effectiveness of SURE is carried out empirically, our results may not be extended to all programs. In the experiments, we use nine projects from two open-source platforms, which are written in different languages (C and Java) and with various functionalities, thus mitigating the threat to an extent. In particular, we use only 30% of the simulated faults to train the model, and use the remaining 70% of the simulated as well as real-world faults to test. This allows us to have higher confidence with respect to the applicability of SURE.

Threats to construct validity relate to the adopted evaluation metrics. We use external metrics (i.e., FMI, JC, PR, and RR) to measure SURE's clustering effectiveness, and use the AP to measure the effectiveness of the failure representation of PMS images. Despite the fact that these metrics have been extensively used by previous works, we consider using more diverse metrics in our future work for further evaluation.

9 Related Work

It is well-recognized that the failure proximity essentially underpins failure indexing techniques. Liu et al. systematically summarized the status of the community of failure proximities, pointing out that CC-based and SD-based strategies can deliver good results [58]. In fact, these two are mainstream tactics in failure indexing to date, and numerous studies have emerged around them in the last two decades.

Let us first focus on the CC-based strategy. As a very early work, Podgurski et al. suggested using execution profiles, including coverage information, as the signature of failures [65]. Since then, this tactic has become popular gradually. For example, Liu et al. formally presented the definition of the CC-based failure proximity, and demonstrated its capability by experiments [58]. Högerle et al. constructed failed test coverage matrices, and used the Weil-Kettler algorithm to rearrange the mentioned coverage matrix into a block diagonal matrix thus achieving failure clustering [32]. Also based on failed test coverage matrices, Steimann and Frenkel utilized two partitioning procedures from integer linear programming to cluster failures, for breaking down the fault localization problem into smaller ones [75]. Huang et al. were concerned with the impact of failure indexing on the effectiveness of multi-fault localization. They conducted an empirical study to explore this topic on the basis of takingCC as the representation of failures [34]. Wu et al. also employed CC as the signature of failures to perform failure clustering. They iteratively chose the cluster of failures with the highest density to conduct single-fault localization, until all faults were fixed [94].

Later, the SD-based strategy, a more sophisticated solution of failure proximity, was proposed by Liu and Han [56], which regards two failures as similar if they suggest roughly the same fault location. Specifically, they used SOBER, a statistical model-based fault localization technique at the predicate granularity [57], to complete the fault location suggestion process. From then on, the SD-based strategy attracted more and more attention from academia. For example, Jones et al. used Tarantula [39], an SBFL technique working at the statement granularity, to suggest finer-grained fault location. As a result, the suspiciousness ranking list that reflects the possibility of each program statement being faulty is utilized to represent failures [38]. Following the workflow of the SD-based strategy, Gao and Wong proposed MSeer, which employed Crosstab [91], an empirically promising SBFL technique, to produce the suspiciousness ranking list that represents failures [27]. MSeer is also the state-of-the-art technique to date in the field of failure indexing. Song et al. investigated the impact of the adopted SBFL techniques on the effectiveness of SD-based failure indexing, concluding that MSeer can be further enhanced by replacing Crosstab with other risk evaluation formulas [73]. Furthermore, they proposed a genetic programming-based approach to automatically evolve formulas solely for failure indexing, and successfully delivered a batch of formulas with promising failure representation effectiveness, such as EFF10-83 [74].

Despite the integration of fault localization techniques, the resource on which the SD-based strategy relies is still only coverage. If failures that are triggered by distinct faults have the same coverage, neither the CC-based nor the SD-based strategy can work well. However, this situation has been demonstrated to be very common in practice, which causes performance degradation in CC and SD strategies. Moreover, the failure representation (CC vectors for the CC-based failure proximity and suspiciousness ranking lists for the SD-based failure proximity) is also tough and time-consuming to comprehend by human developers, which hinders human comprehension of failure indexing results. In this article, we propose SURE, a novel failure indexing technique, to tackle the aforementioned threats to existing approaches. With the support of runtime memory information, SURE remarkably improves the effectiveness of failure indexing, and PMS images, the novel form of failure representation adopted by SURE, help human developers comprehend the result of failure indexing better.

There are some recent works that introduce external profiles to support failure indexing, such as code-independent features in regression testing [29], as well as code features and historical features in continuous integration [4]. We do not consider such types of studies since they go beyond our research scope: 1) Their source information cannot be always available, and 2) This article focuses on failure indexing in the context of multi-fault debugging.

10 Conclusion

In this article, we propose the PM-based failure proximity, and based on that design a novel failure indexing technique, SURE. Experimental results demonstrate the high competitiveness of SURE: it can achieve 101.20% and 41.38% improvements in faults number estimation, as well as 105.20% and 35.53% improvements in clustering, compared with the state-of-the-art technique in this field to date, in simulated and real-world environments, respectively. SURE also provides human developers with insights to better comprehend the failure indexing results. Our human study involving 18 participants shows that the form of failure representation of SURE, i.e., PMS images, can boost human developers' comprehension of failure indexing results by 140.00% and 128.57% compared with the two most advanced and prevalent strategies, while the time cost is reduced by 85.43% and 85.68%, respectively.

In the future, we plan to dig deeper into the characteristics of various forms of failure proximities, and try to combine their advantages for better extracting the signature of failures. A more extensive experiment with a larger scale of benchmarks as well as more diverse evaluation metrics is also being considered.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2009. Spectrum-based multiple fault localization. In *Proceedings of the 24th International Conference on Automated Software Engineering*. 88–99.

[2] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of the 6th International Symposium on Software Reliability Engineering*. 143–151.

[3] Higor Amario de Souza, Marcelo de Souza Lauretto, Fabio Kon, and Marcos Lordello Chaim. 2023. Understanding the use of spectrum-based fault localization. *Journal of Software: Evolution and Process* 36, 6 (2023), e2622.

[4] Gabin An, Juyeon Yoon, Jeongju Sohn, Jingun Hong, Dongwon Hwang, and Shin Yoo. 2022. Automatically identifying shared root causes of test breakages in SAP HANA. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 65–74.

[5] Gabin An, Juyeon Yoon, and Shin Yoo. 2021. Searching for multi-fault programs in Defects4J. In *Proceedings of the 13th International Symposium on Search Based Software Engineering*. 153–158.

[6] 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.

[7] 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.

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

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

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

[11] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1993. Signature verification using a "Siamese" time delay neural network. In *Proceedings of the Advances in Neural Information Processing Systems,* Vol. 6. 737–744.

[12] Dylan Callaghan and Bernd Fischer. 2023. Improving spectrum-based localization of multiple faults by iterative test suite reduction. arXiv:2306.09892. Retrieved from https://doi.org/10.1145/3597926.3598148

[13] Davide Chicco. 2021. Siamese neural networks: An overview. In *Artificial Neural Networks*. John M. Walker (Ed.), Springer, 73–94.

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

[15] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the 2005 Conference on Computer Vision and Pattern Recognition*, Vol. 1. 539–546.

[16] William Dickinson, David Leon, and A. Fodgurski. 2001. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering*. 339–348.

[17] 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.

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

[19] Nicholas DiGiuseppe and James A. Jones. 2012b. Software behavior and failure clustering: An empirical study of fault causality. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*. 191–200.

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

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

[22] 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.

[23] Hugh S. Fairman, Michael H. Brill, and Henry Hemmendinger. 1997. How the CIE 1931 color-matching functions were derived from Wright-Guild data. *Color Research & Application* 22, 1 (1997), 11–23.

[24] Farid Feyzi and Saeed Parsa. 2019. Inforence: Effective fault localization based on information-theoretic analysis and statistical causal inference. *Frontiers of Computer Science* 13 (2019), 735–759.

[25] Wei Fu and Patrick O. Perry. 2020. Estimating the number of clusters using cross-validation. *Journal of Computational and Graphical Statistics* 29, 1 (2020), 162–173.

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

[27] 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.

- [28] Laleh Sh Ghandehari, Yu Lei, Raghu Kacker, Richard Kuhn, Tao Xie, and David Kung. 2018. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering* 46, 6 (2018), 616–645.
- [29] Mojdeh Golagha, Constantin Lehnhoff, Alexander Pretschner, and Hermann Ilmberger. 2019. Failure clustering without coverage. In *Proceedings of the 28th International Symposium on Software Testing and Analysis*. 134–145.
- [30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 2016 Conference on Computer Vision and Pattern Recognition*. 770–778.
- [31] Robert Hirsch. 2005. *Exploring Colour Photography: A Complete Guide*. Laurence King Publishing.
- [32] Wolfgang Högerle, Friedrich Steimann, and Marcus Frenkel. 2014. More debugging in parallel. In *Proceedings of the 25th International Symposium on Software Reliability Engineering*. 133–143.
- [33] Rubing Huang, Dave Towey, Yinyin Xu, Yunan Zhou, and Ning Yang. 2022. Dissimilarity-based test case prioritization through data fusion. *Software: Practice and Experience* 52, 6 (2022), 1352–1377.
- [34] Yanqin Huang, Junhua Wu, Yang Feng, Zhenyu Chen, and Zhihong Zhao. 2013. An empirical study on clustering for isolating bugs in fault localization. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops*. 138–143.
- [35] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. 1999. Data clustering: A review. *Comput. Surveys* 31, 3 (1999), 264–323.
- [36] 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.
- [37] Jiajun Jiang, Yumeng Wang, Junjie Chen, Delin Lv, and Mengjiao Liu. 2023. Variable-based fault localization via enhanced decision tree. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–32.
- [38] 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.
- [39] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering*. 273–282.
- [40] 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*. 467–477.
- [41] Ankur Joshi, Saket Kale, Satish Chandel, and D. Kumar Pal. 2015. Likert scale: Explored and explained. *British Journal of Applied Science & Technology* 7, 4 (2015), 396–403.
- [42] René Just, Darioush Jalali, and Michael D. Ernst. 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*. 437–440.
- [43] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014b. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*. 654–665.
- [44] Leonard Kaufman and Peter J. Rousseeuw. 2009. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons.
- [45] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *Proceedings of the 2017 International Conference on Software Quality, Reliability and Security*. 114–125.
- [46] Maurice George Kendall. 1948. Rank correlation methods. (1948).
- [47] Yunho Kim and Shin Hong. 2022. Learning-based mutant reduction using fine-grained mutation operators. *Software Testing, Verification and Reliability* 32, 7 (2022), e1786.
- [48] Suneel Kumar Kingrani, Mark Levene, and Dell Zhang. 2018. Estimating the number of clusters using diversity. *Artificial Intelligence Research* 7, 1 (2018), 15–22.
- [49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Advances in Neural Information Processing Systems,* Vol. 25. 84–90.
- [50] 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.
- [51] Tien-Duy B. Le, David Lo, and Ferdian Thung. 2015. Should I follow this fault localization tool's output? Automated prediction of fault localization effectiveness. *Empirical Software Engineering* 20 (2015), 1237–1274.
- [52] Shangru Li. 2020. *Research on Fault Location Based on Memory Data*. Master's thesis. Northwestern Polytechnical University.
- [53] Yihao Li and Pan Liu. 2022. A preliminary investigation on the performance of SBFL techniques and distance metrics in parallel fault localization. *IEEE Transactions on Reliability* 71, 2 (2022), 803–817.
- [54] Zheng Li, Yonghao Wu, Haifeng Wang, Xiang Chen, and Yong Liu. 2022. Review of software multiple fault localization approaches. *Chinese Journal of Computers* 45, 2 (02 2022), 256–288.

[55] 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.

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

[57] 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.

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

[59] Muhammad Azhar Mushtaq, Abid Sultan, Muhammad Afrasayab, and Taimoor Zubair. 2019. New cryptographic algorithm using ASCII values and gray code (AGC). In *Proceedings of the Fourth International Conference on Big Data and Computing*. 242–246.

[60] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20, 3 (2011), 1–32.

[61] Farah Naz, Ijaz Ali Shoukat, Rehan Ashraf, Umer Iqbal, and Abdul Rauf. 2020. An ASCII based effective and multi-operation image encryption method. *Multimedia Tools and Applications* 79 (2020), 22107–22129.

[62] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 199–209.

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

[64] 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.

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

[66] 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.

[67] Henrique L Ribeiro, Roberto PA de Araujo, Marcos L Chaim, Higor A de Souza, and Fabio Kon. 2018. Jaguar: A spectrum-based fault localization tool for real-world software. In *Proceedings of the 11th International Conference on Software Testing, Verification and Validation*. 404–409.

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

[69] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556. Retrieved from https://doi.org/10.48550/arXiv.1409.1556

[70] SIR. 2020. The Software Infrastructure Repository. Retrieved from https://sir.csc.ncsu.edu/portal/index.php

[71] Ian Sommerville. 2011. *Software Engineering, 9/E*. Pearson Education India.

[72] Ge Song and Xiaoyang Tan. 2017. Hierarchical deep hashing for image retrieval. *Frontiers of Computer Science* 11 (2017), 253–265.

[73] Yi Song, Xiaoyuan Xie, Quanming Liu, Xihao Zhang, and Xi Wu. 2022a. A comprehensive empirical investigation on failure clustering in parallel debugging. *Journal of Systems and Software* 193 (2022), 111452.

[74] Yi Song, Xiaoyuan Xie, Xihao Zhang, Quanming Liu, and Ruizhi Gao. 2022b. Evolving ranking-based failure proximities for better clustering in fault isolation. In *Proceedings of the 37th International Conference on Automated Software Engineering*. 1–13.

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

[76] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2016. *Introduction to Data Mining*. Pearson Education India.

[77] Zhao Tian, Junjie Chen, and Zhi Jin. 2023. Code difference guided adversarial example generation for deep code models. In *Proceedings of the 38th International Conference on Automated Software Engineering*. 850–862.

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

[79] Béla Vancsics. 2023. *New Algorithms and Benchmarks for Supporting Spectrum-Based Fault Localization*. Ph.D. Dissertation. University of Szeged.

[80] A. Vijayan, T. Gobinath, and M. Saravanakarthikeyan. 2016. ASCII value based encryption system (AVB). *International Journal of Engineering Research and Applications* 6, 4 (2016), 8–11.

[81] MK Vijaymeena and K Kavitha. 2016. A survey on similarity measures in text mining. *Machine Learning and Applications: An International Journal* 3, 2 (2016), 19–28.

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

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

[84] Xingya Wang, Shujuan Jiang, Pengfei Gao, Kai Lu, Bo Lili, Xiaolin Ju, and Yanmei Zhang. 2020. Fuzzy C-means clustering based multi-fault localization. *Chinese Journal of Computers* 43, 2 (2020), 206–232.

[85] Xumeng Wang, Ziliang Wu, Wenqi Huang, Yating Wei, Zhaosong Huang, Mingliang Xu, and Wei Chen. 2023. VIS+AI: Integrating visualization with artificial intelligence for efficient data analysis. *Frontiers of Computer Science* 17, 6 (2023), 176709.

[86] Ming Wen, Zifan Xie, Kaixuan Luo, Xiao Chen, Yibiao Yang, and Hai Jin. 2022. Effective isolation of fault-correlated variables via statistical and mutation analysis. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2053–2068.

[87] Wanzhi Wen. 2012. Software fault localization based on program slicing spectrum. In *Proceedings of the 34th International Conference on Software Engineering*. 1511–1514.

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

[89] 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.

[90] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2011. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61, 1 (2011), 149–169.

[91] W. Eric Wong, Vidroha Debroy, and Dianxiang Xu. 2012. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 3, 42 (2012), 378–396.

[92] 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.

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

[94] 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 (2020), 979–998.

[95] Yan Xiaobo, Liu Bin, and Wang Shihai. 2021. A test restoration method based on genetic algorithm for effective fault localization in multiple-fault programs. *Journal of Systems and Software* 172 (2021), 110861.

[96] J. Xie, Y. Zhou, M. Wang, and W. Jiang. 2017. New criteria for evaluating the validity of clustering. *CAAI Transactions on Intelligent Systems* 12, 6 (2017), 873–882.

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

[98] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering*. 808–819.

[99] 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.

[100] 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.

[101] 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.

[102] Bo Yang, Qian Yu, Huai Liu, Yuze He, and Chao Liu. 2021b. Software debugging analysis based on developer behavior data. *Frontiers of Computer Science* 15, 1 (2021), 151203.

[103] Deheng Yang, Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2021a. Evaluating the usage of fault localization in automated program repair: An empirical study. *Frontiers of Computer Science* 15 (2021), 1–15.

[104] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. 1482–1493.

[105] Shin Yoo. 2012. Evolving human competitive spectra-based fault localisation techniques. In *Proceedings of the 4th International Symposium on Search Based Software Engineering*. 244–258.

[106] Shin Yoo and Mark Harman. 2010. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software* 83, 4 (2010), 689–701.

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

[108] Juyeon Yoon and Shin Yoo. 2021. Enhancing lexical representation of test coverage for failure clustering. In *Proceedings of the 36th International Conference on Automated Software Engineering Workshops*. 232–238.

[109] 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 *Proceedings of the 37th International Conference on Software Engineering*, Vol. 1. 924–935.

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

[111] 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.

[112] 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.

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

Received 2 November 2023; revised 6 April 2024; accepted 26 June 2024