MSeer — an Advanced Technique for Locating Multiple Bugs in Parallel

Ruizhi Gao and W. Eric Wong

Abstract — In practice, a program may contain multiple bugs. The simultaneous presence of these bugs may deteriorate the effectiveness of existing fault-localization techniques to locate program bugs. While it is acceptable to use all failed and successful tests to identify suspicious code for programs with exactly one bug, it is not appropriate to use the same approach for programs with multiple bugs because the due-to relationship between failed tests and underlying bugs cannot be easily identified. One solution is to generate fault-focused clusters by grouping failed tests caused by the same bug into the same clusters. We propose MSeer — an advanced fault localization technique for locating multiple bugs in parallel. Our major contributions include the use of (1) a revised Kendall tau distance to measure the distance between two failed tests, (2) an innovative approach to simultaneously estimate the number of clusters and assign initial medoids to these clusters, and (3) an improved K-medoids clustering algorithm to better identify the due-to relationship between failed tests and their corresponding bugs. Case studies on 840 multiple-bug versions of seven programs suggest that MSeer performs better in terms of effectiveness and efficiency than two other techniques for locating multiple bugs in parallel.

Index Terms — Software fault localization, parallel debugging, multiple bugs, clustering, distance metrics

1 INTRODUCTION

Regardless of the effort spent developing a computer program, it may still contain bugs. In fact, the larger and more complex a program, the higher the likelihood of it containing bugs. When the execution of a test case on a program fails, it suggests that the program has bugs. However, the burden of locating and fixing these bugs is on the programmers. To do so, they must first be able to identify exactly where these bugs are. Known as fault localization, this step can be very time consuming and expensive [59]. Many fault localization techniques use all failed and successful test cases to prioritize code and generate a ranking of statements in descending order of their suspiciousness values. Statements with higher suspiciousness values are pushed towards the top of the ranking, as they are more likely to contain bugs than those at the bottom with lower suspiciousness values. Programmers then examine statements from the top of the ranking to locate the first statement that contains bug(s). Fault localization techniques using this approach have been well reported in the literature [1,2,4,5,17,22,23,33,40,60,62,63,64,65]. Many of these studies assume there is exactly one bug in the program. Since a single-bug assumption may not hold in practice and mixed failed test cases associated with different causative bugs can reduce the effectiveness of a fault localization technique, in this paper we propose to extend existing techniques to debug programs with multiple bugs in parallel.

The major difference between fault localization on programs with multiple bugs and fault localization on programs with exactly one bug is that, for programs with multiple bugs, programmers have to determine the due-to relationship [35] between a failure (and the failed test) and its corresponding bug(s), while, for programs with a single bug, the programmers are not required to do such a determination because there is only one bug in the program.

For programs with exactly one bug, the execution trace collected at runtime can provide information such as whether a statement is covered by a test case. This trace and the corresponding execution result (success or failure) are used by various spectrum-based fault localization techniques to generate a suspiciousness ranking and help programmers locate the bug.

However, it is not appropriate to apply the same process directly to programs with multiple bugs. The major challenge lies in identifying the due-to relationship between failed test cases and the underlying causative bugs. One solution is to produce fault-focused clusters [23] by grouping failed test cases caused by the same bug into the same clusters. That is, failed test cases in the same cluster are related to the same bug, whereas failed test cases in different clusters are related to different bugs. A fault-focused suspiciousness ranking is then generated using failed test cases of a given cluster and some or all of the successful test cases. Examining code along this ranking can help programmers locate the corresponding causative bug linked to this ranking. The key component of this process is obtaining a good clustering on failed test cases. A major challenge is that we do not know the number of bugs in a program in advance. Hence, we are not able to properly determine the number of clusters or assign initial medoids (centers) to these clusters, which are required in order to use many advanced clustering techniques [18,25,38]. Other challenges during clustering include how to properly represent a failed test case, how to measure the difference between two failed test cases (i.e., which distance metric should be used) and how to perform the clustering based on an appropriate
clustering algorithm. Different solutions to these challenges have different impacts on the clustering results, which will accordingly affect the fault localization effectiveness.

We propose an advanced fault localization technique, MSeer, to address all the issues above simultaneously. There are four novel aspects of our work.

First, in MSeer, a failed test case is represented by a suspiciousness ranking of statements generated by a given fault localization technique using the corresponding failed test and all successful tests. The ranking is in descending order of each statement’s likelihood of containing bugs. The advantage of using statements rather than predicates/decisions (as in [35]) and suspiciousness ranking instead of execution trace is explained in Section 2.1.

Second, MSeer uses a revised Kendall tau distance to measure the distance between two failed test cases (i.e., two suspiciousness rankings). Although Kendall tau has been used successfully in other studies such as information retrieval [54] and bioengineering [47], it cannot be applied directly to software fault localization because it assigns the same weight to each statement no matter how suspicious. This is inappropriate and must be modified as suggested by [35]. To correct this problem, we propose giving greater weight to more suspicious statements and smaller weight to less suspicious statements. The rationale behind this modification is explained in Section 2.2.

Third, MSeer applies an innovative approach to estimate the number of clusters and, at the same time, assign initial medoids3 to these clusters (Section 2.3.1). A similar approach was discussed by Yager and Filev [75] and modified by Chiu [8]; It has been used in studies such as the one by Lin et al. for color image segmentation [33]. However, this approach has never been used to help programmers locate software bugs. To do so, we must make two important changes. First, References [75] and [8] only mentioned that the distance between two data points needs to be measured, but they did not specify how to do it. In our study, such distance is measured using the revised Kendall tau in Equation (2) for the reasons provided in Section 2.2. Moreover, instead of using a fixed value for a parameter that is critical in estimating the number of clusters, we propose to use data winsorization to help determine its value. This is necessary as different values should be used in different scenarios to improve the performance. Refer to Section 5.2 for more details.

Fourth, MSeer performs clustering using an improved K-medoids algorithm (Section 2.3.2). Although K-medoids is a popular clustering algorithm and has been used in studies such as graph classification [46], it is too expensive for large data sets because it examines all possible combinations of K data points in order to find appropriate initial medoids. However, since not every combination is used as an initial medoid, we should not be required to examine those combinations which are not used. To address this problem, we propose an improved K-medoids using an innovative approach as described in Section 2.3.1 to help determine appropriate initial medoids without using all possible combinations. In this way, the performance of the original K-medoids can be significantly improved.

The major contribution of this paper is to propose a technique which can effectively locate multiple bugs in parallel. This technique simultaneously uses:

- a revised Kendall tau with a greater weight for more suspicious statements to measure the distance between two failed test cases
- an innovative approach for estimating the number of clusters and determining their initial medoids, with appropriate parameter values decided by data winsorization
- an improved K-medoids algorithm for clustering without examining all possible combinations of K data points

3 In this paper, a medoid refers to a cluster center that belongs to the corresponding cluster.


to solve a very important problem in Software Engineering: how to effectively perform fault localization on a program with multiple bugs.

Case studies on seven medium- to large-sized programs (gzip, grep, make, flex, ant, socat and xmail) written in different languages (C, C++ and Java) with various functionalities were conducted to evaluate the effectiveness and efficiency of MSeer based on several metrics (Section 4.3). In total, 840 multiple-bug faulty versions (2-bug, 3-bug, 4-bug, and 5-bug) of the seven programs were used. A cross-comparison between MSeer and two other techniques (one bug at a time and the second technique proposed by Jones et al. [23]) is reported in Section 4.4. Our results strongly suggest that MSeer performs better in terms of both effectiveness and efficiency.

The remainder of this paper is organized as follows. Section 2 explains our proposed multiple-bug fault localization technique, MSeer, in detail followed by a running example in Section 2.4 to demonstrate how to apply MSeer to localize multiple bugs of a program in parallel. Sections 4 and 5 report our case studies and discuss important aspects related to MSeer. Section 6 presents the threats to validity. Other studies that are related to our technique are presented in Section 7. Our conclusions and direction of future work can be found in Section 8.

2 OUR TECHNIQUE

We now present our proposed technique, MSeer, for effectively locating multiple bugs in a program. We first describe in Section 2.1 how failed test cases are represented and then explain in Section 2.2 a revised Kendall tau distance and the advantage of using it to measure the distances between two suspiciousness rankings. A discussion of how to estimate the number of clusters and their initial medoids appears in Section 2.3.1, followed by a description of our improved K-medoids clustering algorithm in Section 2.3.2. The detailed procedure for MSeer is in Section 2.4.

2.1 Representation of Failed Test Cases

To find the due-to relationship mentioned in Section 1, we propose to use a clustering algorithm, K-medoids, to determine which failed test cases should be used, along with successful test cases, to locate the corresponding causative bug. For a given failed test case, the corresponding representation can be, but is not limited to, one of the following:
a) A statement coverage vector such that an entry 1 at the $i^{th}$ position implies that the corresponding statement $s_i$ is covered by the failed test, and an entry 0 implies it is not.

b) A suspiciousness ranking of statements based on a given fault localization technique using the corresponding failed test and all successful tests.

Representation a) is similar to the representation used to compute T-proximity (trace-proximity) [35] in which failed tests are grouped based on the similarity of their execution traces. Such representation is also used in other multiple-bug fault localization studies [20,22,50]. However, as suggested by Liu et al. [35], this representation is problematic because a fault can be triggered in many different ways. As a result, execution traces of failed test cases due to the same bug can be very different. Excluding some failed test cases only because their statement coverage vectors differ from others and ignoring the fact that the failures caused by all these failed test cases are due to the same bug will reduce the effectiveness of fault localization.

Representation b) is similar to the representation used to compute the R-proximity (rank-proximity), which has been shown to be a better representation than a) in clustering failed tests due to the same bug [35]. In this paper, we use representation b). We also generate suspiciousness rankings in terms of statements rather than predicates. A potential disadvantage of the latter (as explained in [35]) is that if a bug is not in the initial set of predicates selected for examination, additional predicates need to be included via a breadth-first search on the dependence graph of the program being debugged. If too few predicates are selected, enough information for fault localization may not be conveyed, while too many predicates are in themselves a burden for developers to examine. Thus, neither leads to the best result.

2.2 Revised Kendall tau distance

The performance of many clustering algorithms depends critically on a good distance metric over the input space [70]. To better measure the distance between suspiciousness rankings, we propose a revised Kendall tau distance metric. We first introduce the original Kendall tau distance, and then discuss how we improve it by giving greater weight to statements at the top of a suspiciousness ranking. Additional discussion can be found in Section 5.3.

The Kendall tau distance is a metric that counts the number of pairwise disagreements between two rankings of the same size [27]. The larger the distance, the more dissimilar the two rankings are. Given two suspiciousness rankings $\omega$ and $\sigma$, each with $m$ statements, their Kendall tau distance $D(\omega,\sigma)$ is defined as:

$$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j)$$  

- if $(\omega(s_i) - \omega(s_j)) \times (\sigma(s_i) - \sigma(s_j)) < 0$ then $K(s_i, s_j) = 1$
- otherwise $K(s_i, s_j) = 0$

where $\omega(s_i)$ is the position of statement $s_i$ in ranking $\omega$; $s_i$ and $s_j$ constitute a discordant pair of statements if their relative orders in $\omega$ and $\sigma$ disagree. Note that Kendall tau distance can be normalized by dividing by $m(m-1)/2$.

To continue the discussion, let us consider the following three rankings ($r_1$, $r_2$ and $r_3$) with four statements ($s_1$ to $s_4$) in Table 1. The value of each cell gives the position of the statement in the corresponding ranking. Referring to the third column, it shows that statement $s_2$ is at the top (position 1) of rankings $r_1$ and $r_2$ but at position 2 of ranking $r_3$.

<table>
<thead>
<tr>
<th>Statements</th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position in ranking $r_1$</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Position in ranking $r_2$</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Position in ranking $r_3$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

To calculate the Kendall tau distance between $r_1$ and $r_2$, we need to count the number of discordant pairs between statements. For $s_1$ and $s_2$, $K(s_1, s_2) = 0$ because $(r_1(s_1) - r_1(s_2)) \times (r_2(s_1) - r_2(s_2)) = (3 - 1) \times (4 - 1) > 0$. Similarly, we have $K(s_1, s_3) = 0$, $K(s_1, s_4) = 1$, $K(s_2, s_3) = 0$, $K(s_2, s_4) = 0$, and $K(s_3, s_4) = 0$. Hence, $D(r_1, r_2) = \sum_{1 \leq i < j \leq 4} K(s_i, s_j) = 1$. Similarly, $D(r_2, r_3)$ is also 1.

However, we should not give the same weight to all discordant pairs. More precisely, discordant pairs of more suspicious statements (those towards the top of the rankings) contribute more to the distance between two rankings than discordant pairs of less suspicious statements (those at lower positions in the rankings).

Referring to Table 1, the only discordant pair between $r_1$ and $r_2$ is $(s_1, s_3)$ with corresponding statements at the third and fourth positions, while the discordant pair between $r_2$ and $r_3$ is $(s_2, s_3)$ with corresponding statements at the first and second positions. Stated differently, the discordance between $r_2$ and $r_3$ is due to more suspicious statements, whereas the discordance between $r_1$ and $r_2$ is due to less suspicious statements. Therefore, it is reasonable to emphasize that the distance between $r_2$ and $r_3$ should be larger than the distance between $r_1$ and $r_2$. This cannot be accomplished by using the original Kendall tau distance defined in Equation (1), in which $K(s_i, s_j)$ is always one provided that $s_i$ and $s_j$ constitute a discordant pair of statements regardless of their positions in the rankings. In other words, even though $s_i$ and $s_j$ are at very low positions in the rankings, the contribution due to their discordance to the distance is the same as that of two discordant statements at higher positions. To resolve this problem, we propose to assign a greater weight to $K(s_i, s_j)$ while computing $D(r_2, r_3)$ and a smaller weight to $K(s_i, s_j)$ while computing $D(r_1, r_2)$ so that we have $D(r_2, r_3)$ larger than $D(r_1, r_2)$. Thus, we modify the Kendall tau distance by taking into account the position of each statement as follows:

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  

$D(\omega,\sigma) = \sum_{1 \leq i < j \leq m} K(s_i, s_j) \cdot \omega_i \cdot \omega_j$  


during the clustering phase, statements with the same suspiciousness value are sorted according to their statement number. Other techniques such as those reported in [43,50,73] can also be used to break the ties.
where $o(s_i), (s_{i+1}), (s_{i+2})$, (which means they are less suspicious), the smallest $K(s_i, s_j)$ is. Referring to the above example, $D^\prime(r_1, r_2) = o(s_i)^{-1} + o(s_j)^{-1} + o(s_{i+1})^1 + o(s_{i+2})^1 = 1.17$ and $D^\prime(r_2, r_3) = 3$.

### 2.3 Clustering

In data mining, the K-medoids clustering algorithm (hereafter, simply referred to as K-medoids) [25] divides data points into clusters such that members of the same cluster are as similar as possible, and those in different clusters are as dissimilar as possible. It chooses $k$ data points to be the “medoids” and minimizes the distance between data points in a cluster and the medoid of the corresponding cluster. In comparison to the K-means clustering algorithm [18, 38], used in other fault localization studies such as [22], K-medoids has been shown to be very robust to the existence of noise or outliers and generally produces clusters of high quality [25]. Furthermore, the use of K-medoids is more applicable to situations in which the mean of the objects is not defined [48]. This is especially critical in our study, as it is difficult to define the mean of the suspiciousness rankings generated by the failed and the successful test cases.

Density-based spatial clustering of applications with noise (DBSCAN) is a clustering algorithm proposed by Ester et al. [16]. Given a set of data points, it groups points that are closely packed together (i.e., points with many nearby neighbors) and discards points that lie alone in low-density regions. Users have to define two important parameters: (1) the radius of a cluster and (2) the minimum number of points required to form a high-density region to help DBSCAN identify whether a point is in low-density regions. If these two parameters are not set properly, some useful data points will be excluded by DBSCAN. Although this clustering algorithm has been applied to areas such as image processing [21], it is not appropriate for fault localization. This is because failed test cases for some bugs may fall in low-density regions and be excluded during clustering. As a result, we reject critical data points that are essential to helping us locate bugs. Hence, we choose not to use DBSCAN in our study.

One challenge of using either K-medoids or K-means is that we first have to decide the number of clusters and assign the initial medoids (called “centers” in K-means). We will discuss how to estimate the number of clusters and assign the initial medoids in Section 2.3.1. Then, an improved K-medoids is presented in Section 2.3.2 to divide suspiciousness rankings into $K$ clusters so that the program failures caused by executing failed test cases in the same cluster are due to the same bug.

#### 2.3.1 Estimation of the Number of Clusters and Assignment of Initial Medoids

The number of clusters in our study corresponds to the possible number of bugs in a program and also indicates how many fault-focused suspiciousness rankings (generated by failed tests in the same cluster and all the successful tests) we should have for a given debugging iteration. Since, in practice, we do not know how many bugs exist in a program, the biggest challenge to using a clustering-based multiple-bug fault localization technique is how to properly estimate the number of clusters at the beginning. This problem is also reviewed by Hogerle et al. [20].

Overestimating the cluster number will result in generation of redundant fault-focused rankings and require additional effort to locate the same bug, which is counterproductive. Underestimating the cluster number means we do not have an adequate number of fault-focused suspiciousness rankings, which also implies that failed tests in the same cluster may not be due to the same bug.

Furthermore, even if the number of clusters is correct, if the initial medoids are assigned improperly, the clustering result may only converge to a local optimum [52]. Therefore, how to best estimate the number of clusters and how to assign appropriate initial medoids is essential to our technique.

One approach is to set the cluster number to $\sqrt{N_F/2}$ for $N_F$ failed test cases based on the work of Mardia et al. [39]. Another approach is to set the number of clusters to a small percentage of $N_F$ (e.g., if there are 100 failed test cases, the number of clusters can be 5 which is 5% of $N_F$) [13, 22].

Both approaches are based on the number of failed test cases. However, neither can be used in our studies because there is no clear correlation between the number of failed tests and the number of bugs in a program. For example, if a program has 200 failed test cases, using Mardia’s approach we should have 10 (namely, $\sqrt{200/2}$) clusters. But, there is no justification to argue that there are 10 bugs in the program.

In this paper, we use an innovative approach to simultaneously estimate the number of clusters and assign initial medoids to these clusters. A similar approach is discussed by Yager and Filev [75]. They make a grid of virtual data points and compute a potential value for each virtual point based on its distances to the actual data points. A virtual point with many actual points nearby will have a high potential value. Then, the virtual point with the highest potential value is chosen as the first cluster center. The key idea of this algorithm is that once the first cluster center is chosen, the potential of all virtual points is reduced according to their distance from the latest selected cluster center. Virtual points near the first cluster center will have greatly reduced potential. The next cluster center is then placed at the virtual point with the highest remaining potential value. The procedure of acquiring new cluster centers and reducing the potential of surrounding virtual points repeats until the potential values of all virtual points fail below a threshold. This approach is revised by Chiu [8] to further improve its efficiency when used on large-sized data sets. It is used to solve a color image
segmentation problem in a study reported by Lin et al. [33]. However, it has never been applied to the software fault localization domain. To do so, we have to make two critical changes to our scenario.

- Determine how to compute the distance between two data points (namely, two suspiciousness rankings in our study).

The approaches reported in [75] and [8] only suggest that such distance should be measured. However, they did not explain how exactly it should be done. Since there are so many metrics that can be used to compute the distance, their results may be very different and can have critical impacts on the performance of clustering. For example, the Euclidean distance is used by the study in [33]. Although it can be applied to color image segmentation, it is not appropriate for software fault localization. Other metrics like the Hamming distance and the Jaccard distance are not good choices either. Refer to Section 5.3 for more discussion. To solve this problem, we propose to use a revised Kendall tau distance defined in Equation (2) for the reasons explained in Section 2.2.

- Assign an appropriate value to the parameter $\psi$ in Equation (3) for a better estimation of the number of clusters.

Instead of using a fixed value for $\psi$ as Chiu did in his study [8], we propose to use data winsorization to help determine its value. This is necessary as different values should be used in different cases to improve the accuracy of estimation. Refer to Section 5.2 for more details.

What follows describes our approach in detail. For a set of $n$ suspiciousness rankings $\{r_1, r_2, r_3, \ldots, r_n\}$, each generated by one failed test case and all successful tests, the number of clusters can be estimated as follows.

**Step 1:** Measure $D'(r_i, r_j)$ (the revised Kendall tau distance) between rankings $r_i$ and $r_j$ ($1 \leq i, j \leq n$)

**Step 2:** Assign a potential value $P^\theta_r$ for each ranking $r_i$ ($1 \leq i \leq n$) as follows

$$P^\theta_r = \sum_{j=1}^{n} e^{-\alpha D'(r_i, r_j)^2}$$

where $\alpha = 4/\psi^2$. Chiu [8] claims that $\psi$ is sensitive to the actual input space. In our studies, we notice that when $\psi$ is set to half of the 5% winsorized mean of the distance between two distinct rankings, the number of clusters can be better estimated. A winsorized mean [55] is a winsorized statistical measure of central tendency, similar to the mean and median. It involves the calculation of the mean after replacing given parts of a probability distribution or samples at the high and low ends with the most extreme remaining values [55]. For example, given 10 numbers (from $x_1$, the smallest, to $x_{10}$, the largest), the 10% winsorized mean of these numbers is calculated as $(x_1 + x_2 + x_3 + \ldots + x_{10} + x_{10} + x_{10} + x_{10} + x_{10} + x_{10} + x_{10})/10$. The 10% at the low end ($x_1$) and 10% at the high end ($x_{10}$) are replaced by the second smallest $x_2$ and the second largest $x_{10}$, respectively. The objective of winsorization is to reduce the effect of possible outliers, which has been shown to be more effective than data trimming [55]. More details about data winsorization can be found in [14,55].

**Step 3:** After $P^\theta_r$ is computed, we choose the ranking with the highest potential value as $R^\theta$ and set its potential value as $M^\theta$. If there are multiple rankings with the same highest potential value, we randomly choose one ranking to break the tie. If $\theta = 0$, we set $R^\theta$ as the medoid of the first cluster and go to Step 4. The algorithm in Figure 1 is used to determine whether we should add a new cluster with $R^\theta$ as its medoid or end the procedure.

**Step 4:** Use Equation (4) to update the potential value of each ranking $r_i$ ($1 \leq i \leq n$) and then go back to Step 3.

$$P^\theta_{i+1} \leq P^\theta_i - M^\theta \times e^{-\beta D'(r_i, R^\theta)^2}$$

where $\beta = 4/\zeta^2$ and $\zeta = 1.5\psi$. Note that the algorithm guarantees that no ranking will be selected as the medoid for more than one cluster.

<table>
<thead>
<tr>
<th>Stopping criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $M^\theta &gt; \bar{M} M^\theta$ ($\bar{\epsilon} = 0.5$ and $\epsilon = 0.15$ [8])</td>
</tr>
<tr>
<td>Accept $R^\theta$ as a cluster medoid and go to Step 4</td>
</tr>
<tr>
<td>else if $M^\theta &lt; \bar{\epsilon} M^\theta$</td>
</tr>
<tr>
<td>Reject $R^\theta$ and stop</td>
</tr>
<tr>
<td>Let $D'_{\text{min}} = \text{[shortest of the revised Kendall-tau distance between } R^\theta \text{ and all previously found clusters] }$</td>
</tr>
<tr>
<td>if $\frac{D'_{\text{min}}}{\psi} + \frac{M^\theta}{M^\theta} \geq 1$</td>
</tr>
<tr>
<td>Accept $R^\theta$ as a cluster medoid and go to Step 4</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$\bullet$ Reject $R^\theta$ and set the potential value of $M^\theta$ to 0</td>
</tr>
<tr>
<td>$\bullet$ Select the ranking with the next highest potential value as $R^\theta$ and assign its potential value as the new $M^\theta$</td>
</tr>
<tr>
<td>$\bullet$ Repeat the stopping criterion from the beginning</td>
</tr>
</tbody>
</table>

Figure 1. A stopping criterion to determine whether a new cluster should be added.

We use an example of five rankings, each with seven statements, as shown in Table 2, to demonstrate how to estimate the number of clusters and assign the initial medoid to each cluster.

\[ ^6 \text{In the first iteration, } \theta = 0 \text{ and } P^\theta_r \text{ is assigned using Equation (3); for subsequent iterations, } P^\theta_r \text{ is computed using Equation (4).} \]
First, we compute the potential value of each ranking (Steps 1 and 2) for iteration 0. We have $P_1^0 = 1.731$, $P_2^0 = 2.035$, $P_3^0 = 1.796$, $P_4^0 = 1.795$, and $P_5^0 = 1.433$. Based on Step 3, $r_2$ is set as $R^0$ and $M^0 = 2.035$. Since $\theta = 0$, $r_2$ is selected as the first medoid and the procedure proceeds to Step 4.

In Step 4, potential values are updated using Equation (4). We have $P_1^1 = 0.032$, $P_2^1 = 0.0$, $P_3^1 = 1.796$, $P_4^1 = 1.795$, and $P_5^1 = 0.128$ after the first iteration. The procedure then returns to Step 3.

We set $r_3$ as $R^1$ and $M^1 = 1.796$. Since $\frac{\pi M^0}{M^1} = 0.305$ and $\frac{\pi M^0}{M^0} = 1.018$, $M^1$ is larger than $\pi M^0$. Hence, $r_3$ is selected as the second medoid, and the procedure goes to Step 4. The updated potential values after the second iteration are $P_1^2 = 0.032$, $P_2^2 = -8.186$, $P_3^2 = 0.0$, $P_4^2 = 0.174$, and $P_5^2 = 0.128$. After comparing these values, $r_4$ is set as $R^2$ and $M^2 = 0.174$. $M^2$ is now less than $\frac{\pi M^0}{M^0}$, so $r_4$ is rejected as a medoid and the whole procedure ends. In summary, there are two clusters with $r_2$ and $r_3$ as the initial medoids, respectively.

### 2.3.2 Improved K-medoids Clustering Algorithm

K-medoids can be used to divide suspiciousness rankings into $K$ clusters by minimizing the following objective function

$$J_s = \sum_{j=1}^{K} \sum_{i=1}^{Q} D(r_i, c_j)$$

where $r_i$ is a suspiciousness ranking (namely, a data point) in the $j^{th}$ cluster; $c_j$ is the corresponding medoid; $Q$ is the number of rankings in the $j^{th}$ cluster; and $K$ is the total number of clusters.

Referring to Section 2.2, the performance of many clustering algorithms (including K-medoids) depends on which distance metric is used. From the example therein, it is clear that the distance between two suspiciousness rankings should be computed using the revised Kendall tau defined in Equation (2) instead of the original in Equation (1). Problems of using other distance metrics such as Euclidean or Jaccard are discussed in Section 5.3. Hence, in our study the distance is measured using the revised Kendall tau.$^7$

Another significant problem of K-medoids is that during the optimization of Equation (5), we have to examine all possible combinations of $K$ rankings as initial medoids. This can be very expensive because the number of combinations is huge for a large data set. A variant of K-medoids, CLARA [25] has been proposed to deal with large data sets. It draws a small sample from the data set and generates an optimal set of medoids for the sample instead of the entire data set. A problem of this approach is that the quality of the clustering results depends significantly on the sample. As a result, it is not appropriate for our study.

We propose to use the approach described in Section 2.3.1 that can help us determine appropriate initial medoids without examining all the combinations, as described above. Below, we explain the details of the improved K-medoids clustering algorithm.

Let $r_i (1 \leq i \leq n)$ be a suspiciousness ranking, $c_j^\theta (1 \leq j \leq K)$ be the medoid of the $j^{th}$ cluster, and $\theta$ be the number of iterations with an initial value 0 before any iteration. The improved K-medoids for clustering suspiciousness rankings includes the following steps:

**Step 1:** Follow the steps described in Section 2.3.1 to estimate the number of clusters, $K$, and assign an initial medoid $c_j^\theta$ (where $\theta$ equals 0) for each cluster.

**Step 2:** Group rankings into clusters so that the revised Kendall tau distance between a ranking and the medoid of the cluster where it resides is less than or equal to the distance between this ranking and the medoids of other clusters of which it is not a member. If a ranking has the same distance to the medoids of multiple clusters, we randomly assign the ranking to one of these clusters.

**Step 3:** Use Equation (6) to compute the sum of the distance between every ranking and its medoid in each cluster,

$$\text{Sum}_j^\theta = \sum_{i=1}^{Q} D(r_i, c_j^\theta)$$

where $Q$ is the number of rankings in the $j^{th}$ cluster.

**Step 4:** For the $j^{th}$ cluster, choose a ranking different from $c_j^\theta$ as $c_j^{\theta+1}$ and compute $\text{Sum}_j^{\theta+1}$. If there exists at least one ranking that satisfies $\text{Sum}_j^{\theta+1} < \text{Sum}_j^\theta$, then select $c_j^{\theta+1}$ as the new medoid of the $j^{th}$ cluster for the next iteration and go back to Step 2. Otherwise, the clustering terminates.

Let us use the same example as shown in Table 2 to demonstrate how rankings are clustered using the improved K-Medoids. From the example in Section 2.3.1, we conclude there are two clusters (i.e., $K = 2$) with $r_2$ as $c_1^0$ (the medoid of the first cluster during iteration 0) and $r_3$ as $c_2^0$ (the medoid of the second cluster during iteration 0). This implies Step 1 is already complete. We now move on to Step 2. The distance between a ranking and its corresponding medoid is shown in the following table.

<table>
<thead>
<tr>
<th>$D(r_i, c_j^\theta)$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1^0$</td>
<td>2.529</td>
<td>0.0</td>
<td>24.476</td>
<td>23.836</td>
<td>3.967</td>
</tr>
<tr>
<td>$c_2^0$</td>
<td>28.271</td>
<td>24.476</td>
<td>0.0</td>
<td>1.9</td>
<td>20.1</td>
</tr>
</tbody>
</table>

---

$^7$ Hereafter, unless otherwise specified, we use “distance” and “revised Kendall tau distance” interchangeably.
We observe that the distance between \( r_1 \) and \( c_1 \) is less than that between \( r_1 \) and \( c_2 \) (namely, 2.529 < 28.271). The same applies to \( r_2 \) and \( r_3 \). Hence, \( r_1, r_2 \) and \( r_3 \) are grouped together as the first cluster. On the other hand, the distance between \( r_3 \) (and \( r_4 \)) and \( 0_1c \) is larger than that between \( r_3 \) (and \( r_4 \)) and \( 0_2c \) (namely, 24.476 > 23.836 > 1.9 for \( r_4 \)). Hence, \( r_3 \) and \( r_4 \) are in the second cluster. At Step 3, we compute \( \text{Sum}_0 = \sum D(r, c_i) = 6.495 \). If we choose \( r_1 \) as \( c_1 \), the corresponding \( \text{Sum}_0 = 9.090 \) is greater than \( \text{Sum}_1 \). If we choose \( r_3 \) as \( c_1 \), the corresponding \( \text{Sum}_1 = 10.528 \) is also greater than \( \text{Sum}_0 \). Similarly, if we choose \( r_4 \) as \( c_1 \), we have \( \text{Sum}_1 \) equal to \( \text{Sum}_0 \). Therefore, the entire clustering process is complete.

![Figure 2. An overview of MSeer](image)

### 2.4 MSeer: A Technique for Locating Multiple Bugs in Parallel

We now present the details of MSeer – an advanced technique for parallel debugging of multiple bugs in the same program. Figure 2 gives an overview with an explanation of major steps.

- **Step 1: Program execution and data collection**

Execute the program being debugged (\( P \)) against a set of test cases (\( T \)). Collect the statement coverage\(^8\) with respect to each test case. We use \( f_i \) (\( i = 1, 2, 3, \ldots, n \)) to denote \( n \) failed test cases and \( S \) as the set of all successful test cases. If there are no failed test cases, then the debugging is terminated. At this point we can only conclude that no bugs can be revealed by the execution of test cases in \( T \), which does not guarantee that \( P \) contains no bugs.

- **Step 2: Generation of suspiciousness rankings using each failed test case and all successful test cases**

A suspiciousness ranking, \( r_i \) (\( i = 1, 2, 3, \ldots, n \)), is generated by a fault localization technique using \( f_i \) and \( S \).

- **Step 3: Estimation of the number of clusters and assignment of initial medoids**

Follow the approach described in Section 2.3.1 to estimate the number of clusters, \( K \), and assign initial cluster medoids.

- **Step 4: Clustering of suspiciousness rankings**

Use the improved K-medoids described in Section 2.3.2 to cluster all the rankings generated at Step 2 into \( K \) fault-focused clusters.

\(^8\) The execution trace of each test case in terms of other coverage criteria such as predicate can also be collected at this step.
Although it is straightforward to compute the average from the best and worst effectiveness, the converse is not true. Providing the average effectiveness offers no insights on where the best and worst effectiveness may lie and, more importantly, can be ambiguous and misleading. For example, two techniques can have the same average effectiveness, but one has a smaller range between the best and the worst cases while the other has a much wider range. As a result, these two

### 3 A RUNNING EXAMPLE

Let us use a sample program in Figure 3, which takes three integers as input, to demonstrate how MSeer can be used to locate multiple bugs in parallel within the same program. The program is executed on 13 test cases: $t_1 = \{1, 2, 3\}; t_2 = \{2, 3, 4\}; t_3 = \{2, 4, 3\}; t_4 = \{3, 5, 4\}; t_5 = \{4, 6, 5\}; t_6 = \{3, 4, 2\}; t_7 = \{3, 4, 1\}; t_8 = \{4, 3, 2\}; t_9 = \{5, 4, 3\}; t_{10} = \{6, 5, 4\}; t_{11} = \{4, 3, 6\}; t_{12} = \{4, 3, 5\}; t_{13} = \{4, 3, 8\}; five of these ($t_1, t_2, t_6, t_{11}, t_{13}$) cause failures.

At Step 2, we use the Crosstab fault localization technique [66] to generate suspiciousness rankings using each failed test case and all successful test cases. Five rankings so generated are shown in Table 3, where the notation $r_1(t_1)$ indicates that ranking $r_1$ is generated using the failed test case $t_1$. Similarly, $r_2(t_6)$ shows ranking $r_2$ is generated using the failed test case $t_6$, and so on.

We then estimate the number of clusters at Step 3. There are three clusters with $r_2$, $r_3$, and $r_5$ as the initial medoids. An improved K-medoids is used at Step 4 to cluster the five rankings.

At Step 5, once again, we use the Crosstab fault localization technique and the failed test cases in each cluster along with all the successful test cases to generate three fault-focused suspiciousness rankings as shown in Table 4.

Multiple statements may have the same suspiciousness value. As a result, they are tied for the same position in the ranking (see Table 4) at the final stage (Step 6) of locating faulty statements that contain program bugs. This is further explained as follows.

Assume that a number of correct statements have the same suspiciousness as a faulty statement. In the best case, we examine the faulty statement first, in the worst case we examine it last and have to examine all the correct statements with the same suspiciousness, and in the average case we examine some correct statements but not as many as the worst case. This results in three different levels of effectiveness: best, average, and worst.

1. input $a, b, c$; mid = $c$
2. if ($b < c$)
3. if ($a < b$)
4. mid = $c$; //Bug 1 mid = $b$
5. else if ($a < c$)
6. mid = $a$
7. else
8. if ($a > b$)
9. mid = $b$
10. else if ($c < a$)
11. mid = $b$; //Bug 2 mid = $a$
12. case mid of
13. $a$ if ($b \geq 2a$)
14. temp = ($a + b + c)/2$; //Bug 3 ($a + b + c)/3
15. print temp;
16. else print ("a");
17. $b$ if ($a \geq 2a$)
18. temp = $a + b + c$;
19. print temp;
20. else print ("b");
21. $c$ if ($a \geq 2b$)
22. temp = $b * c$;
23. print temp;
24. else print ("c");

Figure 3. A sample program

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
<th>$S_7$</th>
<th>$S_8$</th>
<th>$S_9$</th>
<th>$S_{10}$</th>
<th>$S_{11}$</th>
<th>$S_{12}$</th>
<th>$S_{13}$</th>
<th>$S_{14}$</th>
<th>$S_{15}$</th>
<th>$S_{16}$</th>
<th>$S_{17}$</th>
<th>$S_{18}$</th>
<th>$S_{19}$</th>
<th>$S_{20}$</th>
<th>$S_{21}$</th>
<th>$S_{22}$</th>
<th>$S_{23}$</th>
<th>$S_{24}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1(t_1)$</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>15</td>
<td>16</td>
<td>21</td>
<td>24</td>
<td>22</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>17</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td>23</td>
<td>13</td>
<td>14</td>
<td>19</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$r_2(t_2)$</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>16</td>
<td>17</td>
<td>21</td>
<td>24</td>
<td>22</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>18</td>
<td>10</td>
<td>11</td>
<td>19</td>
<td>23</td>
<td>12</td>
<td>13</td>
<td>20</td>
<td>4</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>$r_3(t_3)$</td>
<td>7</td>
<td>8</td>
<td>22</td>
<td>10</td>
<td>17</td>
<td>18</td>
<td>4</td>
<td>6</td>
<td>23</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>19</td>
<td>11</td>
<td>12</td>
<td>20</td>
<td>5</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>24</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>$r_4(t_4)$</td>
<td>8</td>
<td>9</td>
<td>22</td>
<td>11</td>
<td>16</td>
<td>17</td>
<td>5</td>
<td>7</td>
<td>23</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>18</td>
<td>12</td>
<td>13</td>
<td>19</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>$r_5(t_5)$</td>
<td>6</td>
<td>7</td>
<td>19</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>21</td>
<td>10</td>
<td>11</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>22</td>
<td>12</td>
<td>13</td>
<td>17</td>
<td>23</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

The number in each cell represents the position of the statement in the corresponding ranking.

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
<th>$S_7$</th>
<th>$S_8$</th>
<th>$S_9$</th>
<th>$S_{10}$</th>
<th>$S_{11}$</th>
<th>$S_{12}$</th>
<th>$S_{13}$</th>
<th>$S_{14}$</th>
<th>$S_{15}$</th>
<th>$S_{16}$</th>
<th>$S_{17}$</th>
<th>$S_{18}$</th>
<th>$S_{19}$</th>
<th>$S_{20}$</th>
<th>$S_{21}$</th>
<th>$S_{22}$</th>
<th>$S_{23}$</th>
<th>$S_{24}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_1$</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>16</td>
<td>16</td>
<td>21</td>
<td>24</td>
<td>21</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>16</td>
<td>10</td>
<td>10</td>
<td>16</td>
<td>21</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$F_2$</td>
<td>9</td>
<td>9</td>
<td>22</td>
<td>12</td>
<td>17</td>
<td>17</td>
<td>5</td>
<td>8</td>
<td>22</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>17</td>
<td>12</td>
<td>12</td>
<td>17</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>22</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>$F_3$</td>
<td>6</td>
<td>6</td>
<td>19</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>19</td>
<td>24</td>
<td>19</td>
<td>9</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>19</td>
<td>12</td>
<td>16</td>
<td>19</td>
<td>12</td>
<td>16</td>
<td>12</td>
</tr>
</tbody>
</table>

The number in each cell represents the position of the statement in the corresponding ranking.

Some researchers may claim that instead of using both the best and the worst, we only need to report the average. Although it is straightforward to compute the average from the best and worst effectiveness, the converse is not true. Providing the average effectiveness offers no insights on
techniques should not be viewed as equally effective as suggested by their average effectiveness.

Other researchers may argue it is only necessary to report the worst case. However, programmers will generally not experience the worst case scenario in practice. It is more likely that they will see something between the best and the worst scenarios. Thus, it is better to report the fault localization effectiveness for the best, the average, and the worst cases and perform the cross evaluation under each scenario.

In this example, the best case only has to examine three statements (because $F_1$ has $s_4$ ranked at the first position, $F_2$ has $s_1$ ranked at the first position, and $F_3$ has $s_{10}$ ranked at the first position), while the worst case needs to examine five statements. Two extra statement needs to be examined because (1) both $s_{10}$ and $s_1$ are tied in $F_2$, and a correct statement $s_{10}$ will be examined before the faulty statement $s_1$ is examined; (2) both $s_{10}$ and $s_5$ are tied in $F_3$, and a correct statement $s_5$ will be examined before the faulty statement $s_{10}$ is examined. The average case needs to examine 4 ($\frac{3 + 5}{2}$) statements. A significant point worth noting is that all three bugs can be located in parallel by using the proposed MSeer technique in only one iteration.

4 CASE STUDIES

Section 4.1 provides an overview of the subject programs used in our case studies and data collection. Section 4.2 introduces other techniques used for cross-comparison. Evaluation metrics used in this paper are explained in Section 4.3. Results of case studies are given in Section 4.4.

4.1 Subject Programs and Data Collection

Seven subject programs are used. Five were downloaded from [49]: versions 1.1.2 of gzip (which reduces the size of named files), 2.2 of grep (which searches for a pattern in a file), 3.76.1 of make (which manages building of executables and other products from source code), 1.1 of the flex program (which generates scanners that perform lexical pattern-matching on text), and 1.6 of ant (which builds Java executable files). A set of test cases and faulty versions of each program were also downloaded. Version 1.4.0.2 of the socat program (which establishes two bidirectional byte streams and transfers data between them) and the faulty version CVE-2004-1484 reported in [56] were downloaded from [51], whereas version 1.2.1 of xmail (which serves as a mail server) and the faulty version CVE-2005-2943 reported in [57] were downloaded from [71]. A set of test cases of socat and xmail were also downloaded from [51] and [71], respectively.

In addition to the faulty versions downloaded from the aforementioned websites, additional faulty versions were created using mutation-based fault injection to enlarge our data sets. Studies such as [3,15,34,40] have shown that mutation-based faults can be used to simulate realistic faults and provide reliable and trustworthy results for testing and debugging experiments. In this paper, two classes of mutant operators are used:

- replacement of an arithmetic, relational, logical, increment and decrement, or assignment operator by another operator from the same class
- decision negation in an if or while statement

Studies such as [42,61,62] have reported that test cases that kill mutants generated by relational operator replacement and logical operator replacement are also likely to kill other mutants.

Table 5 gives the size (lines of code), the number of faulty versions, and the number of test cases of each subject program. These programs are either medium or large-sized, written in different languages (gzip, grep, make, flex, and socat in C, ant in Java, and xmail in C++) with various functionalities. Such diversity makes our results more convincing.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>No. of faulty versions</th>
<th>Number of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>6,573</td>
<td>23</td>
<td>211</td>
</tr>
<tr>
<td>grep</td>
<td>12,653</td>
<td>18</td>
<td>470</td>
</tr>
<tr>
<td>make</td>
<td>20,014</td>
<td>29</td>
<td>793</td>
</tr>
<tr>
<td>flex</td>
<td>13,892</td>
<td>22</td>
<td>525</td>
</tr>
<tr>
<td>ant</td>
<td>75,333</td>
<td>23</td>
<td>871</td>
</tr>
<tr>
<td>socat</td>
<td>16,576</td>
<td>15</td>
<td>213</td>
</tr>
<tr>
<td>xmail</td>
<td>50,366</td>
<td>22</td>
<td>314</td>
</tr>
</tbody>
</table>

Faulty versions with multiple bugs were created for each program by injecting bugs from multiple single-bug versions into the same multiple-bug version. For example, a 5-bug version of a program can be created by seeding bugs from five single-bug versions into the program simultaneously. Since there is more than one way to create a 5-bug version, using only one may lead to a biased conclusion. To avoid this bias, 30 distinct faulty versions with 2, 3, 4, and 5 bugs, respectively, for gzip, grep, make, flex, ant, socat, and xmail were randomly created. Altogether, there were 840 multiple-bug programs in our studies. This approach (creating multiple-bug fault versions by injecting bugs from multiple single-bug versions) has been used by many published studies [2,20,22,30 76].

Different tools were used to collect coverage information for each test execution: a revised version of χSuds [72] for C programs, gcov [17] for C++ programs, and Clover [9] for Java programs.

4.2 Two Other Techniques for Cross-Comparison

We compare MSeer to the one-bug-at-a-time (OBA) technique and the second technique proposed by Jones et al. [23]. For each iteration in OBA, all failed test cases (even though they are associated with different causative bugs) and all successful test cases are used in conjunction with a fault localization technique to generate a suspiciousness ranking. Programmers fix the first bug located by the ranking and then execute the modified program against all test cases before moving on to the next iteration. The debugging process terminates when all test cases are executed successfully. Only one bug is fixed in each iteration.

Jones et al. [23] proposed two techniques for parallel debugging of a program with multiple bugs. It is difficult to conduct an experiment using their first technique because the paper did not provide enough detail of how user behavior models were clustered. This point has also been confirmed by
the authors of [2,20]. As a result, only the second technique (hereafter referred to as J2) is used for cross-comparison. J2 first uses Tarantula to generate suspiciousness rankings with respect to each failed test case and all successful test cases. It then measures the Jaccard distance between each pair of rankings and marks two of them as similar if their distance is less than 0.5 (further discussion about why Jaccard distance is not a good candidate for measuring the distance between two rankings can be found in Section 5.3). J2 clusters these rankings by taking a closure of the pairs of rankings that are marked as similar. Fault-focused rankings are generated using all failed test cases in each cluster and all successful test cases. The major differences between MSeer and J2 are:

- J2 uses a simple distance metric (Jaccard), which cannot precisely measure the distance between two suspiciousness rankings (see Section 5.3), and they treat every statement equally without considering the fact that more suspicious statements should contribute more to the distance between two rankings than less suspicious statements (see Section 2.2). A revised Kendall tau distance is proposed in MSeer to overcome these deficiencies.
- J2 applies a hierarchical clustering algorithm, which may not be as effective and efficient as K-medoids [7,26]. Instead, MSeer uses an innovative approach (Section 2.3.1) to estimate the number of clusters and assign initial medoids at the same time. It then uses an improved K-medoids clustering algorithm (2.3.2) to perform the clustering.

For MSeer, the last step of each debugging iteration (Step 6 in Section 2.4) involves fixing the first bug located by each fault-focused suspiciousness ranking. The same applies to J2.

In [23], suspiciousness rankings are generated using Tarantula, a fault localization technique that has been shown to be less effective than Crosstab [66]. For a fair comparison, Crosstab is used as the fault localization technique in all experiments to compare the effectiveness of MSeer, OBA, and J2. The possible impact of different fault localization techniques (D* [64], RBF [65], Ochiai [1], and Tarantula [24]) on our results is discussed in Section 5.1.

4.3 Evaluation of Effectiveness & Efficiency

The effectiveness of multiple-bug fault localization techniques can be measured using one the following metrics.

- **Average number of statements examined**

This metric gives the average number of statements that need to be examined to find all the bugs in a multiple-bug version of a subject program. For discussion purposes, let us assume a program P has n multiple-bug versions. Technique X is more effective in fault localization than technique Y for P, if

\[
\frac{\sum X(i)}{n} \leq \frac{\sum Y(i)}{n},
\]

where X(i) and Y(i) are the number of statements that need be examined to locate all bugs in the ith multiple-bug version of P by X and Y, respectively.

- **T-EXAM score**

The EXAM score used in our previous studies [65,66] gives the percentage of statements that need to be examined until the first bug is located. In this paper, we extend the original EXAM and define a new metric T-EXAM for evaluating the effectiveness of multiple-bug fault localization techniques.

For OBA, if there are μ debugging iterations, the T-EXAM score is defined as:

\[
\text{T-EXAM} = \sum_{i=1}^{\mu} \text{EXAM}_i
\]

where EXAM_i is the percentage of statements that need to be examined to locate the first bug at the ith iteration. Let us use a 3-bug faulty program with 200 statements as an example. Assume that a programmer has to examine 15, 20, and 18 statements in order to find the first bug in the first, second, and third iterations. EXAM_1 equals 100×(15/200) = 7.5, and EXAM_2 and EXAM_3 equal 10 and 9, respectively. Hence, the T-EXAM score for locating all three bugs in this program is 7.5 + 10 + 9 = 26.5.

For MSeer and J2, let μ be the number of debugging iterations and r be the number of fault-focused suspiciousness rankings generated for each iteration. The T-EXAM is defined as:

\[
\text{T-EXAM} = \sum_{i=1}^{\mu} \sum_{j=1}^{r} \text{EXAM}_{ij}
\]

where EXAM_{ij} is the percentage of statements that need to be examined to locate the bug referred to by the jth fault-focused suspiciousness ranking in the ith iteration. For discussion purposes, let us use a 3-bug faulty program with 200 statements as an example. Assume that MSeer needs two iterations to locate all three bugs. In the first iteration, two fault-focused rankings are generated. Following the first ranking, 5 statements need to be examined to locate the bug, whereas 10 statements in the second ranking have to be examined to locate a different bug. In the second iteration, there is one fault-focused ranking which requires an examination of 7 statements to locate the remaining bug. We have EXAM_1,1 = 2.5 (which is 5/200), EXAM_1,2 = 5.0, and EXAM_1,3 = 3.5. Together, the T-EXAM for locating all three bugs is 2.5 + 5.0 + 3.5 = 11.0.

One significant difference between EXAM and T-EXAM is that EXAM gives the percentage of code that needs to be examined to locate the bug in a single-bug program or the first bug of a multiple-bug program [64], whereas T-EXAM is a score that can be used to measure the effectiveness of different techniques to locate all the bugs in a multiple-bug program. The effectiveness of two techniques, X and Y, for debugging multiple-bug programs can be compared based on their T-EXAM scores. If X has a smaller T-EXAM score than Y, then X is considered to be more effective than Y.

- **Wilcoxon Signed-Rank Test**

The Wilcoxon signed-rank test (a.k.a. Mann-Whitney U test) is an alternative to other hypothesis tests such as the paired Student’s t-test and z-test when a normal distribution of the population cannot be assumed) [37] is also used to provide a comparison with a solid statistical basis between the
effectiveness of different techniques. Since we aim to show that MSeer is more effective than another technique, the difference between the number of statements that need to be examined using MSeer and another technique is computed. We evaluate the one-tailed alternative hypothesis that the other techniques require an examination of greater number of statements than MSeer. In this paper, all Wilcoxon signed-rank test are run with Benjamini-Hochberg correction. The null hypothesis is:

\[ H_0: \text{The number of statements examined by other techniques to locate all bugs in a multiple-bug program} \leq \text{the number of statements examined by MSeer.} \]

If \( H_0 \) is rejected (i.e., the alternative hypothesis is accepted), then it implies that MSeer will require an examination of fewer statements than other techniques. This also implies MSeer is more effective than other techniques. Debugging a program using MSeer is independent of debugging the same program using other techniques (J2 or OBA in our case). We assume that the differences in effectiveness using MSeer and the other techniques with respect to a reasonable number of samples (30 in our case) comply with a continuous distribution symmetric about its median.

One way to measure the efficiency of techniques that locate multiple bugs in parallel is to use the number of debugging iterations needed to locate all the bugs in a faulty program. For two techniques X and Y, if the number of debugging iterations required by X is smaller than that required by Y, then X is more efficient than Y. Note that, based on our data, even for programs with more than 50 KLOC and almost 900 test cases, the clustering can still be completed within a few minutes. This suggests that clustering will not be a factor that imposes a significant impact on the efficiency of our proposed technique.

### 4.4 Results

Table 6 to Table 8 give the average number of statements that need to be examined by MSeer, OBA, and J2 with respect to 30 versions of a given program each containing \( \alpha \) bugs (\( \alpha = 2, 3, 4 \) and 5). For example, the average number of statements examined by MSeer with respect to the 3-bug faulty versions of gzip is 41.77 in the best case, 108.17 in the average, and 174.57 in the worst. For OBA, the best is 85.40, the average is 155.45, and the worst is 225.50. For J2, the best is 113.93, the average is 186.55, and the worst is 259.17.

With respect to the 84 scenarios (seven programs with 2, 3, 4, and 5 bugs, respectively, for best, average, and worst cases), we observe that

- MSeer outperforms OBA and J2 in all 84 scenarios. The improvement is very significant in some scenarios, for instance in the average and worst cases of flex, ant and grep.
- Although the effectiveness of J2 and OBA are comparable in most scenarios, J2 is less effective than OBA in some scenarios (such as the best, average, and worst cases of socat and gzip).
- A close examination shows that using the J2 technique may generate redundant fault-focused rankings associated with the same bug. As a result, effort is wasted by locating the same bug more than once, which reduces J2’s effectiveness. Such redundancy rarely happens when the MSeer technique is used.

<table>
<thead>
<tr>
<th>2-bug</th>
<th>gzip</th>
<th>grep</th>
<th>make</th>
<th>flex</th>
<th>ant</th>
<th>socat</th>
<th>xmail</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSeer</td>
<td>12.80</td>
<td>345.77</td>
<td>261.53</td>
<td>11.37</td>
<td>16.73</td>
<td>8.63</td>
<td>215.77</td>
</tr>
<tr>
<td>OBA</td>
<td>28.40</td>
<td>361.57</td>
<td>592.27</td>
<td>33.57</td>
<td>18.93</td>
<td>22.27</td>
<td>473.83</td>
</tr>
<tr>
<td>J2</td>
<td>53.57</td>
<td>368.67</td>
<td>364.50</td>
<td>27.83</td>
<td>23.03</td>
<td>27.83</td>
<td>436.77</td>
</tr>
<tr>
<td>3-bug</td>
<td>MSeer</td>
<td>41.77</td>
<td>492.93</td>
<td>640.70</td>
<td>23.30</td>
<td>35.13</td>
<td>18.40</td>
</tr>
<tr>
<td>OBA</td>
<td>85.40</td>
<td>525.93</td>
<td>888.43</td>
<td>72.43</td>
<td>37.87</td>
<td>37.10</td>
<td>621.90</td>
</tr>
<tr>
<td>J2</td>
<td>113.93</td>
<td>547.40</td>
<td>722.13</td>
<td>63.03</td>
<td>43.43</td>
<td>81.17</td>
<td>570.03</td>
</tr>
<tr>
<td>4-bug</td>
<td>MSeer</td>
<td>70.27</td>
<td>560.03</td>
<td>685.57</td>
<td>67.70</td>
<td>44.77</td>
<td>42.07</td>
</tr>
<tr>
<td>OBA</td>
<td>162.57</td>
<td>589.43</td>
<td>1163.77</td>
<td>111.93</td>
<td>69.57</td>
<td>62.37</td>
<td>930.97</td>
</tr>
<tr>
<td>J2</td>
<td>194.80</td>
<td>614.60</td>
<td>732.60</td>
<td>99.00</td>
<td>75.47</td>
<td>274.97</td>
<td>893.93</td>
</tr>
<tr>
<td>5-bug</td>
<td>MSeer</td>
<td>80.00</td>
<td>598.80</td>
<td>1142.97</td>
<td>104.67</td>
<td>53.40</td>
<td>63.33</td>
</tr>
<tr>
<td>OBA</td>
<td>243.70</td>
<td>649.23</td>
<td>1540.93</td>
<td>170.67</td>
<td>77.17</td>
<td>135.93</td>
<td>1232.73</td>
</tr>
<tr>
<td>J2</td>
<td>275.17</td>
<td>672.63</td>
<td>1212.43</td>
<td>152.37</td>
<td>81.63</td>
<td>340.00</td>
<td>1184.60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2-bug</th>
<th>gzip</th>
<th>grep</th>
<th>make</th>
<th>flex</th>
<th>ant</th>
<th>socat</th>
<th>xmail</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSeer</td>
<td>45.07</td>
<td>498.22</td>
<td>536.95</td>
<td>43.62</td>
<td>39.43</td>
<td>23.37</td>
<td>448.15</td>
</tr>
<tr>
<td>OBA</td>
<td>67.09</td>
<td>805.62</td>
<td>768.07</td>
<td>116.22</td>
<td>89.00</td>
<td>39.30</td>
<td>614.45</td>
</tr>
<tr>
<td>J2</td>
<td>109.12</td>
<td>814.52</td>
<td>641.42</td>
<td>110.02</td>
<td>93.83</td>
<td>43.55</td>
<td>579.27</td>
</tr>
<tr>
<td>3-bug</td>
<td>MSeer</td>
<td>108.17</td>
<td>720.70</td>
<td>885.59</td>
<td>67.35</td>
<td>73.50</td>
<td>49.04</td>
</tr>
<tr>
<td>OBA</td>
<td>155.45</td>
<td>1171.83</td>
<td>1152.12</td>
<td>179.60</td>
<td>177.99</td>
<td>65.47</td>
<td>806.49</td>
</tr>
<tr>
<td>J2</td>
<td>186.55</td>
<td>1194.04</td>
<td>979.15</td>
<td>137.93</td>
<td>183.55</td>
<td>150.65</td>
<td>756.48</td>
</tr>
<tr>
<td>4-bug</td>
<td>MSeer</td>
<td>158.65</td>
<td>1024.55</td>
<td>956.62</td>
<td>104.60</td>
<td>98.24</td>
<td>87.07</td>
</tr>
<tr>
<td>OBA</td>
<td>287.29</td>
<td>1339.73</td>
<td>1361.97</td>
<td>218.03</td>
<td>246.90</td>
<td>102.07</td>
<td>1089.55</td>
</tr>
<tr>
<td>J2</td>
<td>317.67</td>
<td>1362.69</td>
<td>1033.04</td>
<td>192.90</td>
<td>252.07</td>
<td>362.45</td>
<td>1046.95</td>
</tr>
<tr>
<td>5-bug</td>
<td>MSeer</td>
<td>199.14</td>
<td>1121.79</td>
<td>1551.17</td>
<td>150.23</td>
<td>115.94</td>
<td>125.45</td>
</tr>
<tr>
<td>OBA</td>
<td>414.22</td>
<td>1587.88</td>
<td>1801.70</td>
<td>276.80</td>
<td>177.22</td>
<td>182.35</td>
<td>1441.35</td>
</tr>
<tr>
<td>J2</td>
<td>444.59</td>
<td>1610.88</td>
<td>1627.45</td>
<td>247.20</td>
<td>182.60</td>
<td>486.99</td>
<td>1398.77</td>
</tr>
</tbody>
</table>
Next, we present the evaluation using the T-EXAM score. The 3-bug versions of `gzip`, `grep` and `make` in best, average and worst cases are presented in Figure 4. The x-axis is the T-EXAM score while the y-axis is the corresponding percentage of faulty versions with all bugs located. For example, referring to Part (a) of Figure 4, when the T-EXAM score equals 4, 63.33% of 3-bug versions of `gzip` have all their bugs located by MSeer in the best case, 43.33% in the average case, and, 33.33% in the worst case. For OBA and J2, respectively, these percentages are 43.33% and 30.00% (best), 20.00% and 16.67% (average), and 13.33% and 3.33% (worst). The T-EXAM scores in Figure 4 suggest that in most scenarios, MSeer is more effective than OBA and J2, whereas the latter two are comparable in effectiveness to each other. The same observation applies to other programs and faulty versions (2-bug, 3-bug, 4-bug, or 5-bug) even though their curves are not included here due to space limitation.

<table>
<thead>
<tr>
<th></th>
<th>gzip</th>
<th>grep</th>
<th>make</th>
<th>flex</th>
<th>ant</th>
<th>socat</th>
<th>xmail</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-bug</td>
<td>MSeer</td>
<td>77.33</td>
<td>650.67</td>
<td>812.37</td>
<td>75.87</td>
<td>38.10</td>
<td>680.53</td>
</tr>
<tr>
<td></td>
<td>OBA</td>
<td>105.77</td>
<td>1249.67</td>
<td>943.87</td>
<td>198.87</td>
<td>159.07</td>
<td>56.33</td>
</tr>
<tr>
<td></td>
<td>J2</td>
<td>164.67</td>
<td>1260.37</td>
<td>918.33</td>
<td>192.20</td>
<td>164.63</td>
<td>59.27</td>
</tr>
<tr>
<td>3-bug</td>
<td>MSeer</td>
<td>174.57</td>
<td>948.47</td>
<td>1130.47</td>
<td>111.40</td>
<td>79.67</td>
<td>940.93</td>
</tr>
<tr>
<td></td>
<td>OBA</td>
<td>225.50</td>
<td>1817.73</td>
<td>1415.80</td>
<td>286.77</td>
<td>318.10</td>
<td>991.07</td>
</tr>
<tr>
<td></td>
<td>J2</td>
<td>259.17</td>
<td>1840.67</td>
<td>1236.17</td>
<td>212.83</td>
<td>323.67</td>
<td>942.93</td>
</tr>
<tr>
<td>4-bug</td>
<td>MSeer</td>
<td>247.03</td>
<td>1489.07</td>
<td>1227.67</td>
<td>141.50</td>
<td>132.07</td>
<td>1116.23</td>
</tr>
<tr>
<td></td>
<td>OBA</td>
<td>412.00</td>
<td>2090.03</td>
<td>1560.17</td>
<td>324.13</td>
<td>141.77</td>
<td>1248.13</td>
</tr>
<tr>
<td></td>
<td>J2</td>
<td>440.53</td>
<td>2110.77</td>
<td>1333.47</td>
<td>286.80</td>
<td>449.93</td>
<td>1199.97</td>
</tr>
<tr>
<td>5-bug</td>
<td>MSeer</td>
<td>318.27</td>
<td>1644.77</td>
<td>1959.37</td>
<td>195.80</td>
<td>141.77</td>
<td>1321.57</td>
</tr>
<tr>
<td></td>
<td>OBA</td>
<td>584.73</td>
<td>2526.53</td>
<td>2062.47</td>
<td>382.93</td>
<td>477.27</td>
<td>1649.97</td>
</tr>
<tr>
<td></td>
<td>J2</td>
<td>614.00</td>
<td>2549.13</td>
<td>2042.47</td>
<td>342.03</td>
<td>483.57</td>
<td>1612.93</td>
</tr>
</tbody>
</table>

Table 8. Average number of statements examined (worst case)
Figure 4. Effectiveness comparison based on T-EXAM score

(e) average case of grep 3-bug versions

(f) worst case of grep 3-bug versions

(g) best case of make 3-bug versions

(h) average case of make 3-bug versions

(i) worst case of make 3-bug versions

Figure 4. Effectiveness comparison based on T-EXAM score
bugs, a 3-bug program needs three iterations to locate all three
This is consistent with the effectiveness comparison using the
Table 9 to Table 11 give effectiveness comparisons for the
best, average, and worst cases using the Wilcoxon signed-rank
test. Each entry in these tables is the confidence (namely, 1−p-value) with which the alternative hypothesis (MSeer is more effective than other techniques)\(^9\) can be accepted. Based on these results, we make the following observations:

- For grep, flex, and socat, the confidence to accept the
  alternative hypothesis is higher than 99%
- For grep and ant, the confidence to accept the alternative
  hypothesis is at least 90% and much higher in many
  scenarios
- 157 of the 168 scenarios accept the alternative hypothesis
  with a confidence level higher than 90%, and the few
  exceptions still have a confidence in the 80s

Overall, results from the Wilcoxon signed-rank test also
suggest that MSeer is more effective than both OBA and J2.
This is consistent with the effectiveness comparison using the
average number of statements examined and the T-EXAM
score.

We now compare the efficiency of MSeer, J2, and OBA in
terms of the number of iterations. For OBA, the number of
iterations is the same as the number of bugs in the program.
Hence, a 2-bug program requires two iterations to locate both
bugs, a 3-bug program needs three iterations to locate all three
bugs, and so on. For MSeer and J2, this number is actually an
average number. For example, the value 1.17 in the third
column and second row in Table 12 (with a gray background
color) gives the average number of iterations required to
locate both bugs over 30 distinct 2-bug versions of the grep
program. Referring to the explanation at the end of Section
4.1, it is important to use 30 distinct versions instead of just
one version to avoid possible bias.

Table 12. Average number of debugging iterations

<table>
<thead>
<tr>
<th></th>
<th>grep</th>
<th>make</th>
<th>flex</th>
<th>ant</th>
<th>socat</th>
<th>xmail</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-bug MSeer</td>
<td>1.17</td>
<td>1.23</td>
<td>1.47</td>
<td>1.13</td>
<td>1.10</td>
<td>1.27</td>
</tr>
<tr>
<td>J2</td>
<td>1.57</td>
<td>1.63</td>
<td>1.63</td>
<td>1.53</td>
<td>1.47</td>
<td>1.43</td>
</tr>
<tr>
<td>OBA</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3-bug MSeer</td>
<td>1.43</td>
<td>1.53</td>
<td>1.83</td>
<td>1.53</td>
<td>1.27</td>
<td>1.57</td>
</tr>
<tr>
<td>J2</td>
<td>1.90</td>
<td>2.07</td>
<td>1.93</td>
<td>1.86</td>
<td>1.70</td>
<td>2.00</td>
</tr>
<tr>
<td>OBA</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4-bug MSeer</td>
<td>1.93</td>
<td>1.80</td>
<td>2.27</td>
<td>1.93</td>
<td>1.73</td>
<td>1.77</td>
</tr>
<tr>
<td>J2</td>
<td>2.47</td>
<td>2.40</td>
<td>2.40</td>
<td>2.33</td>
<td>2.07</td>
<td>2.20</td>
</tr>
<tr>
<td>OBA</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5-bug MSeer</td>
<td>2.40</td>
<td>2.27</td>
<td>2.57</td>
<td>2.37</td>
<td>2.03</td>
<td>2.10</td>
</tr>
<tr>
<td>J2</td>
<td>2.97</td>
<td>2.93</td>
<td>2.40</td>
<td>2.80</td>
<td>2.47</td>
<td>2.57</td>
</tr>
<tr>
<td>OBA</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Referring to the data in Table 12, we notice that the
average number of debugging iterations required by MSeer
is smaller than that required by J2, which is smaller than the
number of iterations required by OBA in 27 out of 28
scenarios.

---

\(^9\) If we change the alternative hypothesis to “MSeer is more effective than
or as effective as other techniques,” the confidence is 100% for most
scenarios.
The only exception is for the make program where the average number of iterations over 30 distinct 5-bug versions required by MSeer and J2 is 2.57 and 2.40, respectively. Nevertheless, the difference between two numbers is very small (only 0.17). On the other hand, each of them is only about half of the number of iterations required by OBA. A very interesting point worth noting is that even though MSeer has an average number of iterations slightly larger than J2 in this case, the average number of statements examined to locate all five bugs using MSeer is still smaller than that using J2 (referred to Table 6 and Table 8, we have 1,142.97 for MSeer versus 1,212.43 for J2 in the best case, 1,551.17 for MSeer versus 1,627.45 for J2 in the average case, and 1,959.37 for MSeer versus 2,042.47 for J2 in the worst case). With respect to the number of debugging iterations, we also run the Wilcoxon signed-rank test with Benjamini-Hochberg correction by setting the null hypothesis as

\[ H_0: \text{The number of debugging iterations required by other techniques to locate all bugs in a multiple-bug program} \leq \text{the number of debugging iterations required by MSeer}. \]

The confidence to reject the null hypothesis is 100% for all scenarios except for one.

In sum, our data strongly suggest that MSeer is not only more effective (in terms of number of statements examined) but also more efficient (in terms of number of debugging iterations) than both J2 and OBA.

5 DISCUSSION

Some interesting topics related to MSeer are discussed in this section.

5.1 Using Different Fault Localization Techniques

MSeer, OBA, and J2 all require the use of a fault localization technique to generate suspiciousness rankings. We now discuss the possible impact of using different fault localization techniques on their effectiveness. In addition to Crosstab [66], we also use D* (a.k.a. DStar where * equals 3) [64], RBF [65], Ochiai [1], and Tarantula [24]. The average number of statements examined over 30 distinct 4-bug versions of gzip, grep, and make is shown in Table 13.

We make the following observations:

- In all 45 scenarios, MSeer is more effective (examining fewer statements) than OBA, and the difference can be very significant. For example, the average number of statements examined in the best case for gzip is 67.27 for MSeer-D* and 154.43 for OBA-D*. This indicates that MSeer is 129.57% more effective than J2.

- Except for 2 of the 45 scenarios, MSeer is more effective than J2.

When MSeer is more effective, the difference can be very significant. For example, the average number of statements examined in the best case for gzip is 70.27 for MSeer-Crosstab and 194.80 for J2-Crosstab. This indicates that MSeer is 177.22% more effective than J2.

On the other hand, when MSeer is less effective, the difference is small. For example, the average number of statements examined in the best case for make is 912.67 for MSeer-Ochiai and 879.13 for J2-Ochiai. That is, MSeer is only 3.67% less effective than J2. For the other case (best, make, Tarantula), MSeer (996.67) is only 4.44% less effective than J2 (952.37).

- Of the 45 scenarios, OBA is more effective than J2 in 29 scenarios and less effective in 16 scenarios.

- In all but two of the 45 scenarios, MSeer is more effective when Crosstab, D*, and RBF are used to compute the suspiciousness rankings, and less effective when Ochiai and Tarantula are used. The same observation applies to OBA and J2, but without any exceptions (i.e., valid for all 45 scenarios).

### Table 13: Average number of statements examined using MSeer, OBA, and J2 in conjunction with different techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>gzip</th>
<th>grep</th>
<th>make</th>
<th>gzip</th>
<th>grep</th>
<th>make</th>
<th>gzip</th>
<th>grep</th>
<th>make</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSeer-Crosstab</td>
<td>70.27</td>
<td>560.03</td>
<td>685.57</td>
<td>158.65</td>
<td>1024.55</td>
<td>956.62</td>
<td>247.03</td>
<td>1489.07</td>
<td>1227.67</td>
</tr>
<tr>
<td>OBA-Crosstab</td>
<td>162.57</td>
<td>589.43</td>
<td>1163.77</td>
<td>287.29</td>
<td>1339.73</td>
<td>1361.97</td>
<td>412.00</td>
<td>2090.03</td>
<td>1560.17</td>
</tr>
<tr>
<td>J2-Crosstab</td>
<td>194.80</td>
<td>614.60</td>
<td>732.60</td>
<td>317.67</td>
<td>1362.69</td>
<td>1033.04</td>
<td>440.53</td>
<td>2110.77</td>
<td>1333.47</td>
</tr>
<tr>
<td>MSeer-D*</td>
<td>67.27</td>
<td>591.47</td>
<td>677.03</td>
<td>153.17</td>
<td>1084.85</td>
<td>934.78</td>
<td>239.07</td>
<td>1578.23</td>
<td>1192.53</td>
</tr>
<tr>
<td>OBA-D*</td>
<td>154.43</td>
<td>665.83</td>
<td>1105.57</td>
<td>272.92</td>
<td>1336.13</td>
<td>1293.87</td>
<td>391.40</td>
<td>2006.43</td>
<td>1482.17</td>
</tr>
<tr>
<td>J2-D*</td>
<td>185.07</td>
<td>690.93</td>
<td>695.97</td>
<td>301.79</td>
<td>1347.63</td>
<td>981.39</td>
<td>418.50</td>
<td>2005.23</td>
<td>1266.80</td>
</tr>
<tr>
<td>MSeer-RBF</td>
<td>92.07</td>
<td>553.27</td>
<td>704.87</td>
<td>182.05</td>
<td>1020.44</td>
<td>956.62</td>
<td>247.03</td>
<td>1489.07</td>
<td>1289.73</td>
</tr>
<tr>
<td>OBA-RBF</td>
<td>178.83</td>
<td>648.37</td>
<td>1280.13</td>
<td>326.32</td>
<td>1525.95</td>
<td>1537.17</td>
<td>473.80</td>
<td>2043.53</td>
<td>1794.20</td>
</tr>
<tr>
<td>J2-RBF</td>
<td>214.27</td>
<td>670.67</td>
<td>805.87</td>
<td>360.44</td>
<td>1551.72</td>
<td>1169.67</td>
<td>506.60</td>
<td>2427.37</td>
<td>1533.47</td>
</tr>
<tr>
<td>MSeer-Ochiai</td>
<td>102.63</td>
<td>608.33</td>
<td>912.67</td>
<td>196.83</td>
<td>1075.20</td>
<td>1220.29</td>
<td>291.03</td>
<td>1542.07</td>
<td>1527.90</td>
</tr>
<tr>
<td>OBA-Ochiai</td>
<td>195.07</td>
<td>707.33</td>
<td>1396.53</td>
<td>344.74</td>
<td>1607.68</td>
<td>1634.37</td>
<td>494.40</td>
<td>2508.03</td>
<td>1872.20</td>
</tr>
<tr>
<td>J2-Ochiai</td>
<td>233.77</td>
<td>737.57</td>
<td>879.13</td>
<td>381.20</td>
<td>1635.23</td>
<td>1239.65</td>
<td>528.63</td>
<td>2532.93</td>
<td>1600.16</td>
</tr>
<tr>
<td>MSeer-Tarantula</td>
<td>115.63</td>
<td>702.07</td>
<td>996.67</td>
<td>212.48</td>
<td>1195.67</td>
<td>1299.67</td>
<td>309.33</td>
<td>1689.23</td>
<td>1602.67</td>
</tr>
<tr>
<td>OBA-Tarantula</td>
<td>211.33</td>
<td>766.27</td>
<td>1512.90</td>
<td>373.47</td>
<td>1741.65</td>
<td>1770.57</td>
<td>535.60</td>
<td>2717.03</td>
<td>2028.23</td>
</tr>
<tr>
<td>J2-Tarantula</td>
<td>253.23</td>
<td>798.97</td>
<td>952.37</td>
<td>412.95</td>
<td>1771.49</td>
<td>1342.94</td>
<td>572.67</td>
<td>2744.00</td>
<td>1733.50</td>
</tr>
</tbody>
</table>
5.2 Apply MSeer to Programs with a Single Bug

As a multiple-bug fault localization technique, MSeer should also be effective on programs with a single bug. We compare the effectiveness of MSeer with that of Crosstab using all seven programs (gzip, grep, make, flex, ant, socat, and xmail) and their faulty versions in Table 5. Table 14 gives the average number of statements examined over all faulty versions of each program.

Table 14. Average number of statements examined using MSeer and Crosstab on programs with a single bug

<table>
<thead>
<tr>
<th></th>
<th>MSeer</th>
<th>Crosstab</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>grep</td>
<td>117.79</td>
<td>128.14</td>
</tr>
<tr>
<td>make</td>
<td>213.75</td>
<td>312.32</td>
</tr>
<tr>
<td>flex</td>
<td>10.51</td>
<td>12.28</td>
</tr>
<tr>
<td>ant</td>
<td>16.78</td>
<td>27.70</td>
</tr>
<tr>
<td>socat</td>
<td>10.25</td>
<td>15.64</td>
</tr>
<tr>
<td>xmail</td>
<td>136.00</td>
<td>225.47</td>
</tr>
</tbody>
</table>

With respect to the 21 scenarios (7 subject programs for best, average, and worst cases), we observe that MSeer has the same effectiveness as Crosstab in 15 scenarios and is only slightly less effective than Crosstab in 6 scenarios by examining a few more statements. We choose Crosstab for comparison because studies such as [64,66] have shown that it is a very effective fault localization technique for programs with a single bug. Therefore, if MSeer is as effective as Crosstab, it clearly suggests that MSeer can also be applied to single-bug programs.

5.3 Distance Metrics

Using a good distance metric is critical to the performance of clustering. To investigate whether the revised Kendall tau distance performs better than the original Kendall tau distance, we compare the effectiveness of MSeer using these two metrics. Table 15 shows the average number of statements examined over 30 distinct 4-bug versions of gzip, grep, and make, where MSeer-RK employs the revised Kendall tau distance and MSeer-OK uses the original Kendall tau distance.

Table 15. Average number of statements examined using MSeer-RK and MSeer-OK

<table>
<thead>
<tr>
<th></th>
<th>MSeer-RK</th>
<th>MSeer-OK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>gzip</td>
<td>70.27</td>
<td>158.65</td>
</tr>
<tr>
<td>grep</td>
<td>560.03</td>
<td>1024.55</td>
</tr>
<tr>
<td>make</td>
<td>685.57</td>
<td>956.62</td>
</tr>
</tbody>
</table>

We observe that the average number of statements examined in the best case for gzip is 70.27 for MSeer-RK and 90.57 for MSeer-OK. The increase in effectiveness is 28.89%, which is very significant. A similar observation also applies to another eight scenarios. These results clearly indicate that we should use the revised Kendall tau distance.

In [23], Jones et al. used the Jaccard distance metric to measure the distance between two rankings. This is inappropriate for the reasons explained below. Given two rankings $r_i$ and $r_j$, the Jaccard distance between them is defined as:

$$\text{Jaccard}(r_i, r_j) = 1 - \frac{|r_i \cap r_j|}{|r_i \cup r_j|}$$

(7)

where $|r_i \cap r_j|$ is the size of the intersection of $r_i$ and $r_j$, and $|r_i \cup r_j|$ is the size of the union of $r_i$ and $r_j$. First of all, the Jaccard distance metric only works on part of a ranking instead of the entire ranking. Otherwise, set $r_i$ is the same as set $r_j$ (except that statements may be ranked at different positions), and the Jaccard distance between any two rankings in our studies will always be zero. Second, if we only consider statements at the top $\delta \%$ of two rankings, the Jaccard distance may not represent the true distance between these two rankings. For example, let ranking $r_1=\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$ and ranking $r_2=\{s_1, s_3, s_2, s_1, s_8, s_5, s_6, s_7\}$, and use only the top 50% of statements in each ranking to compute the Jaccard distance between $r_1$ and $r_2$. As a result, the intersection and the union of the top 50% statements of $r_1$ and $r_2$ are identical to each other. Hence, the Jaccard distance so computed is zero, which implies that there is no distance between $r_1$ and $r_2$. This conclusion is clearly very questionable. The fundamental problem is that the discordance between statements is not considered while computing the distance between two suspiciousness rankings.

Other metrics such as the Hamming distance and the Euclidean distance more strongly emphasize the difference between the same positions of two data vectors. However, with respect to a suspiciousness ranking used for fault localization, the relative order between statements is the most important attribute. Hence, while clustering suspiciousness rankings to determine the failed tests that are due to the causative bug (Step 4 in Section 2.4), it is more appropriate to use a revised Kendall tau distance as described by Equation (2) in Section 2.2.

Spearman distance [77] is another metric that can be used to measure the distance between two rankings. However, Kendall and Gibbons [27] have pointed out that Spearman is much more sensitive to errors and discrepancies in data and is less reliable and less interpretable than Kendall tau. Hence, we use the revised Kendall tau distance in our study.

5.4 The Importance of Estimating the Number of Clusters and Assigning Initial Medoids

As discussed in Section 2.3.1, overestimating the number of clusters will result in generation of redundant fault-focused rankings and expending unnecessary effort to locate the same bug more than once. On the other hand, underestimating the number of clusters gives us fewer fault-focused suspiciousness rankings with an adverse consequence such that failed tests in the same cluster may not be due to the same bug.

Our approach can estimate the number of clusters and assign the initial medoids simultaneously. A possible alternative is to set the number of clusters to $\sqrt{N_f/2}$ for $N_f$ failed test cases and randomly select initial medoids as suggested by Mardia et al. [39]. We compare the effectiveness of MSeer using these two approaches over 30 distinct 4-bug versions of gzip, grep, and make, and present the results in
Table 16, where MSeer-Gao uses our approach and MSeer-Mardia follows the suggestion in [39].

<table>
<thead>
<tr>
<th></th>
<th>MSeer-Estimation</th>
<th>MSeer-Mardia</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>gzip</td>
<td>70.27</td>
<td>158.65</td>
</tr>
<tr>
<td>grep</td>
<td>560.03</td>
<td>1024.55</td>
</tr>
<tr>
<td>make</td>
<td>685.57</td>
<td>956.62</td>
</tr>
</tbody>
</table>

We observe that the average number of statements examined in the best case for gzip is 70.27 for MSeer-Gao and 316.22 for MSeer-Mardia. The increase in effectiveness is 350.01%, which is very significant. A similar observation also applies to another eight scenarios. These results clearly indicate that it is critical that MSeer should use our approach to estimate the number of clusters and assign the initial medoids.

6 THREATS TO VALIDITY

Given that the evaluation of the effectiveness of MSeer was conducted empirically, our results may not be extended to all programs. However, we took steps to counter this threat by employing seven programs (either medium- or large-sized) written in three different languages (C, C++ and Java) with various functionalities to make our evaluation more comprehensive. In addition to the faulty versions that were downloaded together with the subject programs, we created additional faulty versions with mutation-based faults (Section 4.1) to augment and enlarge the data set. This allows us to have higher confidence with respect to both the applicability of MSeer to different programs and its efficacy in providing fault localization. Appropriate caution may be needed while interpreting our results due to the application of mutation-based faults.

Studies such as Parnin and Orso [45] argued that programmers will only examine the first few statements in a suspicious ranking. If bug locations are not found at this point, other approaches must be applied to help programmers continuously search the locations that contain bugs. Under this scenario, the effectiveness will be computed differently.

While the metrics described in Section 4.3 are suitable for measuring the effectiveness and efficiency of fault localization techniques, by themselves they do not provide a complete picture of the effort spent in locating bugs, as developers may not examine statements one at a time and may not spend the same amount of time examining different statements. We assume that if a developer examines a faulty statement, he or she will identify the corresponding fault(s). By the same token, we assume that a developer will not identify a non-faulty statement as faulty. If this does not hold in practice, then the examination effort may increase. We also assume that once a bug is located, it can be fixed without introducing new bugs. These assumptions are common to all fault localization studies discussed in this paper and those reported in a comprehensive survey published in 2016 [60].

A potential threat of using the Wilcoxon signed-rank test to show that MSeer is more effective (in terms of the number of statements examined) than OBA and J2 is that we assume the underlying data points (i.e., differences in effectiveness from the 30 faulty versions) follow a continuous distribution symmetric about its median.

7 RELATED STUDIES

In this section, we provide an overview of related studies in addition to those discussed in the preceding sections, directing readers interested in further details to the accompanying references.

Fault localization techniques based on suspicious rankings [1,2,4,5,18,23,24,34,41,61,63,64,65,66] have been well reported in the literature. A Survey on Software Fault Localization [60] presents a comprehensive review of the current fault localization state-of-art.

Researchers have reported studies on combining multiple fault localization techniques for a better effectiveness. For example, Lucia et al. [36] leveraged the diversity of multiple existing spectrum-based fault localization techniques to better localize bugs using data fusion methods. Xuan and Monperrus [74] proposed MULTRIC, a learning-based approach to combining multiple fault localization techniques. Debroy and Wong used a consensus-based strategy to improve the quality of fault localization [19].

Program invariants are also used in fault localization studies. Le et al. [31] proposed a fault localization technique that employs a learning-to-rank strategy, using likely program invariants and suspiciousness scores as features, to rank program methods based on their likelihood of being a root cause of a failure.

A popular but unrealistic assumption is that multiple bugs in the same program behave independently. Debroy and Wong [10] examined interferences that may take place between bugs, and they found that such interferences may manifest themselves to either trigger or mask execution failures. Results based on their experiments indicate that destructive interference (when execution fails due to a bug but no longer fails when another bug is added to the same program) is more common than constructive interference (when execution fails in the presence of two bugs in the same program but does not in the presence of either bug alone) because failures are masked more often than triggered by additional bugs. It is possible that a program with multiple bugs suffers from both destructive and constructive interferences. Di Giuseppe and Jones [12] also reported that multiple bugs have an adverse impact on the effectiveness of spectrum-based techniques. Similar observations have also been found in [76]. The existence of different types of interferences between multiple bugs may result in imperfect clustering such that some failed tests associated with a given bug are incorrectly grouped with failed tests responsible for another bug. However, based on our data, MSeer is sufficiently robust to absorb such noise yet still be effective in locating multiple bugs in parallel.

OBA has been adopted in studies using the DStar technique [64] and a fault localization technique based on a Bayesian reasoning framework [2]. A potential weakness of most techniques based on Bayesian reasoning (e.g., [3,28]), including the technique in [2], is that they all assume program components fail independently; in other words, interferences between multiple bugs as described in [10,12,76] are ignored.
This assumption in general does not hold in practice. In addition, although the technique described in [2] may be helpful for localizing individual bugs, it is less suitable for parallel debugging.

Other studies have also used the approach of clustering failed test cases. Podgurski et al. [44] applied supervised and unsupervised pattern classifications as well as multivariate visualization to execution profiles of failed test cases with an approach for estimating the number of clusters in order to group them into fault-focusing clusters. Steimann and Frenkel [20,50] used the Weil-Kettler algorithm, an integer linear programming technique, to cluster failed test cases. However, both studies perform clustering of failed tests based on their execution profiles and suffer from the weakness discussed in Section 2.1 (Representation (a)). A better approach is to use suspiciousness rankings to cluster failed test cases (Representation (b)).

Yu et al. [76] proposed a technique to separate failed tests that only executed a single bug from failed tests that executed multiple bugs by comparing the *slam-distance* between each pair of the rankings generated using a failed test and all successful tests. No case studies were conducted to show how their technique could improve the fault localization effectiveness.

Lamraoui and Nakajima [30] proposed a fault localization technique based on a new program encoding, *full flow-sensitive trace formula*. The proposed technique is evaluated on single-bug and multiple-bug faulty versions of a small-sized program *teas* with 173 lines of code. This technique still follows the OBA technique, which is different from the focus of our study, as MSeer emphasizes locating multiple bugs simultaneously.

Most studies in software fault localization focus on proposing new techniques to identify suspicious locations that may contain bugs and examining their effectiveness. However, results from these studies are only approximate, and their correctness is not guaranteed. Locations so identified still need to be verified while bugs are actually fixed. Parnin and Orso [45] conducted a study to compare the actual performance of developers in debugging with and without using a fault localization technique. They claimed that several assumptions made by existing fault localization techniques do not hold in practice because programmers will only examine the first few statements in a suspicious ranking. Xie et al. [69] reported that fault localization techniques might even slightly weaken programmers’ abilities in identifying the root faults. On the other hand, Xia et al. [67] suggested that the studies [45] and [69] suffer from several drawbacks: (1) only using small-sized programs, (2) only involving students, and (3) only using dated fault localization techniques. In response, Xia et al. conducted a study based on four large-sized open source projects with professional software programmers. They found that fault localization techniques could help professionals reduce their debugging time, and the improvements were statistically significant and substantial. To investigate how fault localization should be improved to better benefit practitioners, Kochhar et al. [29] highlighted some directions by conducting a literature review.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose MSeer, an advanced fault localization technique for debugging multiple bugs in parallel. As described in Figure 2, MSeer first generates one suspiciousness ranking for each failed test case and then groups these rankings into fault-focused clusters in which all failed test cases in the same cluster are associated with the same causative bug. In the clustering phase, MSeer applies a revised Kendall tau distance (Equation (2) in Section 2.2) to measure the distance between each pair of suspiciousness rankings; an innovative approach for simultaneously estimating the number of clusters and assigning initial medoids (Section 2.3.1); and an improved K-medoids clustering algorithm (Section 2.3.2) to perform clustering. All failed tests in each fault-focused cluster are then used with all successful test cases to generate fault-focused suspiciousness rankings. Examining these fault-focused rankings leads to parallel localization of different bugs.

Empirical data from our case studies (See Section 4) using seven programs, *gzip, grep, make, flex, ant, socat*, and *xmail* (either medium-sized or large-sized), in three different languages (C, C++ or Java) with various functionalities indicates that MSeer outperforms two other techniques, OBA and J2, in terms of both fault localization effectiveness and efficiency.

Studies that target a wider range of industrial programs with real bugs are currently in progress to further validate the effectiveness and efficiency of MSeer. In the future, we plan to further explore the impacts of using different distance matrices and clustering algorithms on the performance of MSeer.

REFERENCES

Artificial Intelligence, pp. 152–158 Washington D.C., USA, July 2013


49. The Software Infrastructure Repository (http://sir.unl.edu/portal/index.html, accessed August 2016)


70. XMail (http://www.xmailserver.org/, accessed August 2013)


This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2776912, IEEE Transactions on Software Engineering
