Fusing Fault Localizers

Lucia[1], David Lo[1], and Xin Xia[2]
[1]School of Information Systems, Singapore Management University
[2]College of Computer Science and Technology, Zhejiang University
{lucia.2009,davidlo}@smu.edu.sg,xxkidd@zju.edu.cn

ABSTRACT

Many spectrum-based fault localization techniques have been proposed to measure how likely each program element is the root cause of a program failure. For various bugs, the best technique to localize the bugs may differ due to the characteristics of the buggy programs and their program spectra. In this paper, we leverage the diversity of existing spectrum-based fault localization techniques to better localize bugs using data fusion methods. Our proposed approach consists of three steps: score normalization, technique selection, and data fusion. We investigate two score normalization methods, two technique selection methods, and five data fusion methods resulting in twenty variants of *Fusion Localizer*. Our approach is bug specific in which the set of techniques to be fused are adaptively selected for each buggy program based on its spectra. Also, it requires no training data, i.e., execution traces of the past buggy programs.

We evaluate our approach on a common benchmark dataset and a dataset consisting of real bugs from three medium to large programs. Our evaluation demonstrates that our approach can significantly improve the effectiveness of existing state-of-the-art fault localization techniques. Compared to these state-of-the-art techniques, the best variants of *Fusion Localizer* can statistically significantly reduce the amount of code to be inspected to find all bugs. Our best variants can increase the proportion of bugs localized when developers only inspect the top 10% most suspicious program elements by more than 10% and increase the number of bugs that can be successfully localized when developers only inspect up to 10 program blocks by more than 20%.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging Aids*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Corrections*

Keywords: Fault Localization; Data Fusion

1. INTRODUCTION

Many fault localization techniques have been proposed to locate the root cause of a program failure by analyzing the program traces (i.e., the abstraction of program behaviors). Spectrum-based fault localization techniques [3, 6, 18, 22, 24, 48] compare the spectra of correct and failed executions to identify program elements (i.e., statements, blocks, methods, and components) that are likely to be the root cause of a program failure. These techniques often use statistical analysis to assign a suspiciousness score to each program element based on its likelihood to be faulty. The higher the score, the more suspicious an element is. A list of the most suspicious program elements is then presented to developers. Developers can then inspect the list starting from the most suspicious elements. Lucia et al. have found that the best performing spectrum-based fault localization techniques vary for different buggy programs [26, 27]. Some techniques can rank faulty elements at the top positions for some buggy programs, while for other buggy programs, they rank faulty elements low in the list.

In this work, we aim to better localize the bugs by leveraging diversity of existing spectrum-based fault localization techniques (in particular 40 association measures [27], Tarantula [18], and Ochiai [3]). Since these techniques are lightweight, we could inexpensively obtain suspiciousness scores for program elements by using different techniques. Different from the approach proposed by Santelices et al. [36] that analyze several types of program spectra using a single technique, we combine many techniques that analyze a single type of program spectra (i.e., block hit spectra).

Data fusion methods have been proposed in the domain of information retrieval to rank documents such that the most relevant ones are in the top positions by combining ranking information from different retrieval systems [42]. We incorporate data fusion methods with the goal of ranking the faulty program elements higher in the list by combining the scores or ranks assigned to program elements by different fault localization techniques. Our approach, referred to as `Fusion Localizer`, normalizes the suspiciousness scores of different fault localization techniques, selects fault localization techniques to be fused, and combines the selected techniques using a data fusion method. We propose 20 variants of `Fusion Localizer` that use different score normalization, technique selection, and data fusion methods.

Related to our work, Wang et al. [39] propose an approach that linearly combines the scores of a number of association measures to rank faulty program elements at the top positions. Their approach generates a weight for each association measure using genetic algorithm based on a set of training data (i.e., program traces of the past program failures and the corresponding faults of the past failures). However, training data is often unavailable in many cases such as when developers work on a new software or when developers do not store traces of the past program failures. Furthermore, the bugs and failures in the training set may not be representative enough for the incoming buggy programs, which could limit the effectiveness

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642983.

of the approach in localizing faults. Different from this technique, our approach requires no training set to leverage diversity of fault localization techniques. Furthermore, our approach is bug specific – it adaptively chooses a different set of techniques to be fused for different buggy programs based on their program spectra.

We have evaluated our approach on a commonly used benchmark dataset consisting of 10 small to medium sized programs written in C and Java, and our own dataset consisting of 30 real bugs collected from 3 larger programs – namely Rhino, Lucene, and Ant. We compare our approach against a number of state-of-the-art spectrum-based fault localization techniques that also do not require training data and analyze a single type of program spectra, i.e., Tarantula [17, 18], Ochiai [3], theoretically best spectrum-based fault localization techniques [44], and theoretically best genetic programming (GP) based fault localization techniques [45]. Our experiment results show that our approach outperforms these techniques. The best performing variants of *Fusion Localizer* (i.e., $F_{CombANZ}^{Zero-One,Bias}$ and $F_{CombANZ}^{Zero-One,Overlap}$) statistically significantly outperform the state-of-the-art spectrum-based fault localization techniques. These best variants require statistically significantly smaller average percentage of code inspected to locate faulty elements as compared to the best performing state-of-the-art fault localization techniques (i.e., 21% vs. 24%). When developers only inspect 10% of the most suspicious program elements, these best variants could improve the percentage of bugs localized by the best performing state-of-the-art fault localization techniques by 11% to 26%. Furthermore when developers only inspect the top 10 most suspicious program blocks, these best variants could improve the number of bugs localized by the best performing state-of-the-art fault localization techniques by 23% to 26%.

The main contributions of this work are as follow:

1. We leverage diversity of 40 association measures [27], Tarantula [18], and Ochiai [3] to better localize bugs using data fusion methods.
2. We provide a bug specific approach which adaptively selects a set of techniques to be fused for each buggy program.
3. We propose an approach that does not require any training data (e.g., program traces of the past program failures) to select which techniques to be fused to better localize bugs in programs.
4. We show that combining the lists of most suspicious program elements recommended by different fault localization techniques (i.e., association measures, Tarantula, and Ochiai) using data fusion methods can improve the effectiveness in localizing faults and statistically significantly outperform the state-of-the-art spectrum-based fault localization techniques.

We organize this work as follows. We first present closely related work in Section 2. Section 3 provides a motivating example to illustrate the benefit of our data fusion approach. We elaborate our approach in Section 4. Section 5 presents our experiment results that demonstrate the effectiveness of our approach. We finally conclude and mention future work in Section 6.

2. RELATED WORK

A number of fault localization techniques have been proposed to help developers find the locations of faults in programs [3, 7, 12, 12, 14, 17, 18, 20, 21, 24, 26, 27, 32–34, 49, 50]. In this section, we discuss several techniques that are closely related to this work. The survey here is by no means complete.

One family of fault localization techniques is spectrum-based fault localization techniques that analyze program spectra of correct and failed executions using statistical analysis to identify possible faulty program elements (e.g., statements, basic blocks, functions, and components) [3, 17, 18, 26, 27]. Jones and Harrold propose a technique called *Tarantula* to measure the suspiciousness of program elements to be the root cause of program failures based on the spectra of correct and failed executions of the programs, and then descendingly rank the elements based on their suspiciousness scores [17, 18]. Similarly, Abreu et al. propose a technique called Ochiai similarity coefficient, which is well-known in the Biology community, to localize faults, and show that Ochiai can effectively localize bugs [3]. A number of association measures introduced in statistic and data mining community have also been applied for fault localization [26, 27]. The studies in [26, 27] show that there is no single measure that outperforms other measures for all bugs. Our technique aims to leverage diversity of the above techniques to better localize bugs.

Instead of empirically studying the effectiveness of a fault localization technique on various faults, Naish et al. theoretically analyze a number of formulas that can be used to compute suspiciousness scores of program elements in "idealized conditions" [28]. Their study is later extended by Xie et al. which theoretically analyze 30 formulas and group them into equivalence classes [44]. They prove that, under some conditions, two families (i.e., ER1 and and ER5) outperform the rest. These families include five formulas: Naish1, Naish2, Wong1, Russel & Rao, and Binary.

Instead of manually designing fault localization formulas, Yoo generates a number of formulas using Genetic Programming [47]. The effectiveness of these formulas are then theoretically studied by Xie et al. [45]. They find that GP13,GP02, GP03, and GP19 are the best ones for fault localization. Instead of generating new formulas, we combine multiple formulas to improve their effectiveness in localizing faults.

The above techniques [3, 17, 18, 26, 27, 45, 47] analyze a single type of program spectra to assign suspiciousness scores to program elements. Differently, Santelices et al. collect *a number of program spectra types* (i.e., statement, branches, and du-pair spectra), where for each program spectra, *a single fault localization technique* (i.e., Ochiai) is used to assign a suspiciousness score for each program element [36]. For each program element, they combine its suspiciousness scores produced by analyzing different types of program spectra. Different from Santelices et al.'s technique, our technique analyzes *a single program spectra type* (i.e., block spectra) using *a number of fault localization techniques*. Thus, we are investigating orthogonal directions and it is possible to combine our approach and Santelices et al.'s approach in a future work.

Our technique is also related to the technique proposed by Wang et al. [39]. For every program element, their technique linearly combines the scores of a number of association measures where each measure obtains a weight which is calculated based on a set of training data using genetic algorithm. Different from their technique, our technique does not require any training data which makes our technique applicable to new programs or when program spectra of relevant past program failures are unavailable. Also, their technique applies the same set of measures and the same set of weights to localize bugs for different buggy programs. Different from Wang et al.'s technique, our technique selects a different set of techniques or set of weights to localize faults for each buggy program based on its program spectra. While Wang et al.'s approach is one-size-fits-all, our approach is bug specific.

3. MOTIVATING EXAMPLE

This section provides an illustration of the benefit of using data fusion to leverage different techniques to help ranking faulty program elements at the top positions. Figure 1 contains a sample program code, excerpted from one of our subject programs, i.e.,

Block ID	Program Elements	Suspicious Scores			Normalized Scores			CombSUM
		Ochiai	Klosgen	Piatetsky Shapiro	Ochiai	Klosgen	Piatetsky Shapiro	
1	double a, x; double ap, del, sum; int n; double temp; if (x <= 0.0)	0.82	0.31	-0.04	0.79	0.9	0.75	2.44
2	{return 0.0;}	0.39	0.06	0	0	0	1	1
3	del = sum = 1.0 / (ap = a); for (n = 1; n <= ITMAX; ++n){	0.93	0.34	-0.15	1	1	0	2
4	sum += del *= x / ++ap; if (Abs(del) < Abs(sum) * EPS){	0.93	0.34	-0.15	1	1	0	2
5	/*BUGS: supposed to be:*/ /*temp = sum * exp(-x + a*log(x)-LGamma(a))*/ temp = sum * exp(x + a * log(x) - LGamma(a)); return temp; }}	0.93	0.34	0	1	1	1	3

Figure 1: An Example of Using a Data Fusion Method

method $gser$ of tot_info version 8. In this example, we divide the code into five program blocks. The bug is in Block 5 where the value assigned to a variable $temp$ is incorrectly calculated.

Various spectrum-based fault localization techniques, such as Ochiai [3], Klosgen [27], and Piatetsky Shapiro [27], can be used to assign a suspiciousness score to each program block. Based on the score of each program block, developers could inspect the program blocks starting from the most suspicious blocks. Figure 1 also shows the suspiciousness score of each program block as output by Ochiai, Klosgen, and Piatetsky Shapiro. These scores are computed by analyzing the program traces collected during the executions of the test cases that come with the program (i.e., tot_info).[1]

According to Ochiai and Klosgen, the most suspicious program blocks are Blocks 3 to 5, followed by Block 1, then Block 2. Since Blocks 3 to 5 receive the same score, in the worst case, developers need to inspect three blocks to find the buggy block. According to Piatetsky Shapiro, Blocks 2 and 5 are the most suspicious program blocks, followed by Blocks 3, 4, and 1. Since Blocks 2 and 5 receive the same score, in the worst case, developers need to inspect two blocks to find the buggy block. In this example, locating the faulty block based on the recommendation by Piatetsky Shapiro requires examining fewer blocks than when following the recommendation by Ochiai or Klosgen.

A simple data fusion method to combine a set of techniques is CombSUM [10, 11]. CombSUM can be used to combine the scores output by Ochiai, Klosgen, and Piatetsky Shapiro to produce a new score for each program block. For each program block, the new score is calculated by summing up the normalized scores given by Ochiai, Klosgen, and Piatetsky Shapiro. As the ranges of the scores produced by these techniques may be different, we need to normalize the scores to make them comparable.[2] From Figure 1, the CombSUM score for Block 5 is 1 + 1 + 1 = 3. We also compute the CombSUM score for the other blocks similarly. Finally, we find that Block 5 is the most suspicious block as it has the highest score among all of the blocks. Using the list of most suspicious blocks recommended by CombSUM, developers could locate the faulty block by only inspecting the first block, which is more effective than using the list of most suspicious blocks recommended by either Ochiai, Klosgen, or Piatetsky Shapiro. In this example, we show that a data fusion method can be used to boost the effectiveness of fault localization techniques.

[1]Please refer to [3, 27] for the formulas that these techniques use to compute the suspiciousness scores.

[2]We elaborate the details of the normalization process in Section 4.

4. FUSION LOCALIZER

In this section, we present our approach that incorporates data fusion methods to locate the source of a program failure. The overview of this approach is presented in Section 4.1. Three main steps of our approach namely score normalization, technique selection, and data fusion are elaborated in Sections 4.2, 4.3, and 4.4 respectively. For each step, we present several techniques that we can use to achieve the respective goal.

4.1 Overview

Our approach combines the scores from different spectrum-based fault localization techniques to produce a new ranking with the goal of improving fault localization effectiveness. In this work, we employ spectrum-based fault localization techniques because they are lightweight and have good accuracies. Nevertheless, it is also possible to use other fault localization techniques that can assign scores to program elements. The overall framework of our approach named *Fusion Localizer* is shown in Figure 2.

The input to our approach is a set of spectrum-based fault localization techniques which computes the suspiciousness scores of all program elements in a buggy program. In this work, we use two well-known spectrum-based fault localization techniques: Tarantula [17, 18] and Ochiai [3], as well as 40 association measures that have been studied for fault localization [26, 27].

Different techniques use different formulas to calculate suspiciousness scores. Each formula has its own characteristics, especially in terms of the range and distribution of the suspiciousness scores computed using it. Thus, normalizing the scores is essential so that scores produced by different measures can be compared with one another. For every technique, the suspiciousness scores that are assigned to program elements can be normalized into a new set of scores that fall in the range of zero to one. Section 4.2 elaborates methods that we use in this work to normalize the scores.

After the scores are normalized, our approach adaptively selects techniques to be fused together based on the spectra of the buggy program. Our goal is to select a set of techniques that complement one another well for a particular buggy program. Section 4.3 elaborates the methods that we use in this work to select the techniques.

Given a set of normalized scores from the selected techniques, we combine the scores using a number of existing data fusion methods. A new set of scores would then be assigned to program elements for the given buggy program. These new scores can be used to create a new list for developer's inspection. Section 4.4 elab-

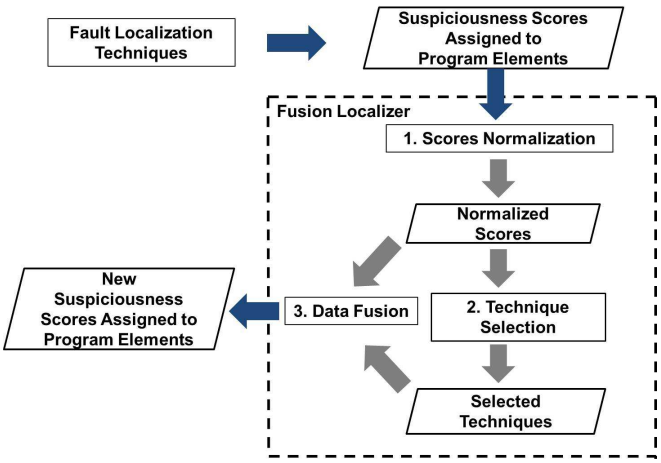

Figure 2: Overview of Fusion Localizer

orates the methods that we use to combine the normalized scores from the selected fault localization techniques.

4.2 Step 1: Score Normalization

In this work, we apply two score normalization methods: Zero-One score normalization [42] and Reciprocal ranking normalization [23, 42]. Each of them can be used as the first step of *Fusion Localizer*. The following paragraphs discuss how these normalization methods work.

1) Zero-One Score Normalization: This method transforms scores from different fault localization techniques into the same range i.e., zero to one. The method works as follows.

Let n be the total number of techniques and m be the total number of program elements for a given buggy program, then $s_i(e_j)$ denotes the score of the j-th program element, assigned by the i-th technique, where $1 \leq j \leq m$ and $1 \leq i \leq n$. Furthermore, let max_s_i denotes the maximum score produced by the i-th technique, and min_s_i denotes the minimum score produced by the i-th technique. The normalized score of the j-th program element given by the i-th technique is calculated as follows:

$$s\_norm_i(e_j) = \frac{s_i(e_j) - min\_s_i}{max\_s_i - min\_s_i}$$

2) Reciprocal Rank Normalization: Instead of directly normalizing a list of scores produced by different techniques, this method considers the ranks of program elements and transforms them into normalized scores. The method works as follows.

Let n be the total number of techniques and m be the total number of program elements in a buggy program. Also, let $r_i(e_j)$ denotes the rank of the j-th program element, assigned by the i-th technique, where $1 \leq j \leq m$ and $1 \leq i \leq n$. The rank of a program element is the number of program elements with higher or the same scores. The normalized score of the j-th program element ranked by the i-th technique is calculated as follows:

$$s\_norm_i(e_j) = \frac{1}{r_i(e_j)}$$

4.3 Step 2: Technique Selection

In this work, we adapt overlap-based selection [42] and bias-based selection [30] to select a subset of techniques to be fused together from the input set of fault localization techniques. The adapted methods are instance (or bug) specific since the selected techniques may differ for different buggy programs. Furthermore, these methods do not require any training data to select techniques. Each of them can be used as the second step of *Fusion Localizer*. The following paragraphs elaborate how we adapt these two methods for fault localization.

Table 1: Example: Overlap Based Selection

Technique	Top 5 Most Suspicious Blocks
$Ochiai$	Block 2, Block 3, Block 4, Block 7, Block 8
$Klosgen$	Block 4, Block 5, Block 6, Block 7, Block 9
$Piatetsky\ Shapiro$	Block 1, Block 4, Block 5, Block 6, Block 8
$Tarantula$	Block 4, Block 5, Block 6, Block 8, Block 10

1) Overlap-Based Selection: This method selects techniques to be fused together based on the overlap between the list of top-K most suspicious program elements produced by a fault localization technique and the top-K lists produced by other techniques. By default, we set K to be 10% of the total number of program elements that the input buggy program has. If this number is less than 10, we set K to 10. This method first measures the *overlap rate* of each technique with other techniques as follows:

DEFINITION 4.1 (OVERLAP RATE). *Let L_{all} be a set of program elements that appears in at least one of the top-K lists produced by the set of fault localization techniques. Also, let L_i be a set of program elements that appears in the top-K list of the i-th technique but not in the top-K lists of other techniques. The overlap rate the i-th technique with other techniques is calculated as follows:*

$$o\_rate_i = \frac{L_{all} - L_i}{L_{all}}$$

Example: Given a buggy program, consider four fault localization techniques (i.e., Ochiai [3], Klosgen [27], Piatetsky Shapiro [27], and Tarantula [17, 18]) that return the top 5 most suspicious program elements (i.e., program blocks) as shown in Table 1.

Based on Table 1, the set of program blocks returned by at least one of the four techniques, denoted by L_{all}, is {Block 1, Block 2, Block 3, Block 4, Block 5, Block 6, Block 7, Block 8, Block 9, Block 10}. Ochiai recommends two program blocks that are not recommended by the other techniques, i.e., L_{Ochiai}={Block 2, Block 3}. As for Klosgen, Piatetsky Shapiro, and Tarantula, each of them recommends one block that is not recommended by other techniques – $L_{Klosgen}$={Block 9}, $L_{Piatetsky\ Shapiro}$={Block 1}, and $L_{Tarantula}$={Block 10}. Hence, the overlap rate of Ochiai among the other techniques can be calculated as $\frac{(10-2)}{10} = 0.8$. We can compute the overlap rate of the other three techniques in the same way– they are $\frac{(10-1)}{10} = 0.9$.

After calculating the overlap rate of each technique, we sort the techniques in ascending order of their overlap rates and select the top-N techniques. In this way, we include techniques that have more unique results (i.e., top-K lists). By default, we set N to be 50% of all input techniques.

2) Bias-Based Selection: This method selects a subset of techniques to be fused based on the bias rate of each technique towards the *norm* considering the top-K lists returned by each technique. By default, we set K to be 10% of the total number of program elements that the input buggy program has. If this number is less than 10, we set K to 10.

This method represents each technique as a vector of zeroes and ones representing whether each of the program elements occurs in its top-K list. It constructs the *norm* which is a vector containing the number of top-K lists each program element belongs to. This method then computes a bias rate for each technique as follows:

DEFINITION 4.2 (BIAS RATE). *Let n be the number of input techniques and m be the number of program elements that appear in a top-K list produced by at least one of the techniques. Let L_i be a vector of zeros and ones that represents whether each program element appears in the top-K list of the i-th technique, where $1 \leq i \leq n$. The norm L_{all} is a vector containing the number of top-K lists each program element appears in. The bias rate of the i-th technique is calculated as follows:*

$$Bias(L_i, L_{all}) = 1 - Sim(L_i, L_{all})$$

$$Sim(L_i, L_{all}) = \frac{\sum_{j=1}^{m} L_j \times L_{all_j}}{\sqrt{\sum_{j=1}^{m} L_j^2} \times \sqrt{\sum_{j=1}^{m} L_{all_j}^2}}$$

In the above formulas, Sim(L_i, L_{all}) is the cosine similarity between vector L_i and vector L_{all} [46].

Example: Based on Table 1, the set of program blocks that appears in at least one top-5 lists is {Block 1, Block 2, Block 3, Block 4, Block 5, Block 6, Block 7, Block 8, Block 9, Block 10}. Block 4 is recommended by four techniques, while Blocks 5, 6, and 8 are recommended by three techniques. Block 7 is recommended by two techniques, and the rest of the blocks are recommended by only one technique. From these pieces of information, the norm L_{all} is {1, 1, 1, 4, 3, 3, 2, 3, 1, 1}. To calculate the bias rate of Ochiai to the norm, i.e., Bias(L_{Ochiai}, L_{all}), we first calculate the similarity score of Ochiai with the norm. The similarity score of L_{Ochiai} with the norm L_{all} is 0.4824 which is calculated as follows:

$$Sim(L_1, L_{all}) = \frac{(1+1+4+2+3)}{\sqrt{10 \times 52}}$$

Based on the similarity score, the bias rate of Ochiai is 1 - 0.4824 = 0.5176. We can compute the bias rate of the other techniques in the same way. The bias rate of Klosgen is $1 - \frac{13}{\sqrt{10 \times 52}}$ =0.43. Piatetsky Shapiro has the same bias rate as Tarantula, i.e., $1 - \frac{14}{\sqrt{10 \times 52}}$ =0.386.

For each technique, we calculate the bias rate of the results and sort them in decreasing order of their bias rates. We then select the top-N of the techniques, with the aim of including techniques that are less similar towards the norm. By default, we set N to be 50% of the input fault localization techniques.

4.4 Step 3: Data Fusion

Our approach adapts an unsupervised data fusion method proposed in the information retrieval community to combine normalized scores from selected fault localization techniques. In this work, we leverage five well-known unsupervised data fusion methods in the domain of information retrieval, namely CombSUM [10, 11], CombMNZ [10, 11], CombANZ [10, 11], correlation-based fusion methods [42], and Borda count [4]. Each of them can be used as the third step of *Fusion Localizer*.

The following paragraphs elaborate how the five popular fusion methods can be adapted for fault localization. We use the example shown in Figure 1 to illustrate how each method works.

1) CombSUM: This method combines the scores from different techniques, by simply summing up their scores. This method assumes that each technique is equally important.

Example. Based on Figure 1, the set of new scores of Blocks 1 to 5 would be {2.44, 1, 2, 2, and 3}.

2) CombANZ: This method combines the scores from different techniques, by computing the average of the non-zero scores. Let e_j denotes the j-th program element and T_i denotes the i-th technique. The score assigned to program element e_j by technique T_i is denoted as $T_i(e_j)$. Suppose there are n techniques and m_{e_j} denotes the number of techniques that assign a non-zero score to e_j, CombANZ calculates the new score for e_j as follows:

$$Score(e_j) = 1/m_{e_j} \times \sum_{i=1}^{n} T_i(e_j)$$

Example. Based on Figure 1, the set of new scores of Blocks 1 to 5 would be $\{\frac{2.44}{3}, \frac{1}{1}, \frac{2}{2}, \frac{2}{2}, \frac{3}{3}\}$={0.81, 1, 1, 1, 1}.

3) CombMNZ: CombMNZ is a variant of CombANZ; it multiplies the summation of all scores for a given element with the number of techniques that assign a non-zero score to the element. Let e_j denotes the j-th program element and T_i denotes the i-th technique. The score assigned to program element e_j by technique T_i is denoted as $T_i(e_j)$. Suppose there are n techniques and m_{e_j} denotes the number of systems that give a non-zero score to e_j, CombMNZ calculates the new score for e_j as follows:

$$Score(e_j) = m_{e_j} \times \sum_{i=1}^{n} T_i(e_j)$$

Example. Based on Figure 1, the set of new scores of Blocks 1 to 5 would be $\{2.44 \times 3, 1 \times 1, 2 \times 2, 2 \times 2, 3 \times 3\}$ = {0.732, 1, 4, 4, 9}.

4) Correlation-based methods: Different from the above methods, correlation-based methods assume each technique is not equally important. The importance of a technique is represented by its weight. Based on the weights of the techniques, these methods linearly combine (i.e., sum up) the scores assigned by different techniques multiplied by their weights. Let e_j denotes the j-th program element and $T_i(e_j)$ denotes the score assigned to program element e_j by the i-th technique. Also, let w_i denotes the weight assigned to the i-th technique and n be the number of techniques. The new score of program element e_j is calculated as follows:

$$Score(e_j) = \sum_{i=1}^{n} w_i \times T_i(e_j) \quad (1)$$

The following paragraphs describe the weight calculation procedures of two correlation-based methods: CorrA and CorrB.

4.1) CorrA: This method computes the correlation among the techniques based on the overlap of the lists of top-N most suspicious program elements returned by the techniques. The method aims to minimize the domination of a certain group of techniques that tend to return similar results, by assigning a heavier weight to a technique that has less correlation with other techniques.

Let the sets of top-N most suspicious program elements returned by the i-th and j-th techniques be denoted as L_i and L_j respectively, where $i \neq j$. The *overlap ratio* between L_i and L_j is calculated as follows:

$$OverlapRatio_{ij} = 2 \times \frac{|L_i \bigcap L_j|}{2 \times N}$$

Suppose, there are n techniques, we calculate the weight of the i-th technique, based on the average of its overlap ratio with other techniques as follows:

$$w_i = 1 - \frac{1}{n-1} \sum_{j=1,2,..,j \neq i}^{n} OverlapRatio_{ij}$$

Example. Based on the example in Table 1 and the top 3 most suspicious program elements returned by each technique, the overlap ratio between Ochiai and Klosgen is 1, while the ratio between Ochiai and Piatetsky Shapiro is 0.33. Thus, the weight assigned to Ochiai is 1-$\frac{1.33}{2}$ =0.335. Similarly, the weights for Klosgen and Piatetsky Shapiro are 0.335 and 0.67. Based on these weights, we calculate the set of new scores for these blocks using Equation 1 – the set of new scores would be {1.07, 0.67, 0.67, 0.67, 1.34}.

In this work, we denote the variant of CorrA which calculates the correlation among the measures based on the top 10% of the most suspicious program elements by $CorrA\_Top10\%$ and another variant that is based on the top 50% of the most suspicious program elements as $CorrA\_Top50\%$.

4.2) CorrB: CorrB is a correlation-based method that also computes the weight of a technique based on the overlap of its list of top-N most suspicious program elements with other lists produced by other techniques. Let the list of top-N most suspicious program elements returned by the i-th technique be denoted as L_i. For a program element e, let us denote the number of top-N lists it belongs to as $list(e)$. Let L_{all} denotes a set of program elements which appears in at least one of the lists produced by the techniques. Suppose there are n techniques to be fused, the weight of the i-th technique can be calculated as follows:

$$w_i = 1 - \frac{(\sum_{e \in L_i} list(e)) - |L_i|}{|L_i| \times |L_{all}|}$$

where $|L_i|$ denotes the number of elements returned by the i-th technique and $|L_{all}|$ denotes the number of elements that appear in at least one of the lists returned by the techniques.

Example. Based on the example in Table 1 and the top 3 most suspicious program elements returned by each technique, L_{Ochiai}, $L_{Klosgen}$, and $L_{Piatetsky\ Shapiro}$ are {Block 3, Block 4, Block 5}, {Block 3, Block 4, Block 5}, and {Block 2, Block 1, Block 5} respectively. Thus, L_{all} is {Block 1, Block 2, Block 3, Block 4, Block 5}. Also, $list(Block3)$, $list(Block4)$, and $list(Block5)$ are 2, 2, and 3 respectively. From the above, the weight assigned to Ochiai is 1-$\frac{7-3}{3\times5}$ = 0.73. The weights for Klosgen and Piatetsky Shapiro can be computed in the same way and they are 0.73 and 0.87 respectively. Therefore, the set of new scores for these blocks calculated using Equation 1 would be {1.87, 0.87, 1.46, 1.46, 2.33}.

In this work, we denote the variant of CorrB method which calculates the correlation among the measures based on the top 10% of the most suspicious program elements by $CorrB\_Top10\%$ and another variant of CorrB method which is based on the top 50% of the most suspicious program elements as $CorrB\_Top50\%$

5) Borda count: This method converts the normalized scores that are assigned to program elements by each selected technique into ranks – program elements with higher scores would obtain smaller ranks. For each element, this method sums up the *ranking points* of an element given by a set of techniques. The ranking point of an element given by a technique is defined as the substraction of the element's rank in the list produced by the technique from the total number of program elements in the input buggy program.

Let e_j denotes the j-th program element and $r_i(e_j)$ denotes the rank assigned to program element e_j by the i-th technique. Also, let N_e denotes the number of program elements and n denotes the number of techniques. Borda count calculates the new score for program element e_j as follows:

$$Score(e_j) = \sum_{i=1}^{n} (N_e - r_i(e_j))$$

Table 2: Example of Ranks and Ranking Points Given by Ochiai, Klosgen, and Piatetsky Shapiro

Block	Ranks	Ranking Points
Block 1	{4, 4, 3}	{1, 1, 2}
Block 2	{5, 5, 2}	{0, 0, 3}
Block 3	{3, 3, 5}	{2, 2, 0}
Block 4	{3, 3, 5}	{2, 2, 0}
Block 5	{3, 3, 2}	{2, 2, 3}

Example. Table 2 shows the ranking points for each program block in Figure 1 given by Ochiai, Klosgen, and Piatetsky Shapiro. Based on the summation of their ranking points, the set of new scores for Blocks 1 to 5 would be {4, 3, 4, 4, 7}.

5. EMPIRICAL EVALUATION

This section presents our subject programs, our evaluation metrics, and our experiment results. Some threats to validity are also discussed.

5.1 Dataset

We evaluate the effectiveness of data fusion methods presented in Section 4.4 to localize faults in 230 buggy programs. Each buggy program contains a single bug that could span across multiple program elements. Table 3 briefly describes our subject programs.

In this work, we evaluate eight subject programs written in C: one real program, namely space, from the Software-artifact Infrastructure Repository (SIR) [37] and seven Siemens programs [16, 37], namely print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. Each of the eight programs has a number of buggy versions. We manually instrument these buggy programs at block level. After excluding buggy versions that our instrumentation cannot reach (e.g., versions with bugs in global variable declarations), we evaluate 154 buggy versions.

We also analyze two real Java programs from the SIR [9] namely: NanoXML [36] and XML-Security [36]. In this work, we analyze four versions of NanoXML (i.e., versions 1, 2, 3, and 5) and three versions of XML-Security (i.e., versions 1, 2, and 3). Each program version has multiple buggy versions and we also instrument them at block level. After excluding buggy versions that have no failed test cases, we evaluate 46 buggy versions.

The above subject programs have often been used to evaluate many fault localization techniques [3, 17, 18, 26, 27]. Despite their popularity, the size of the above subject programs are relatively small. Most of the bugs in the above subject programs are also synthetic rather than real. Thus, we create another dataset consisting of 30 real bugs from three larger programs namely Rhino [8], Lucene [2], and Ant [1]. Rhino bugs and test suite are obtained from the iBugs repository which was created by Dallmeier and

Table 3: Dataset Descriptions

Dataset	LOC	Number of Buggy Versions	Number of Test Cases
print_token	478	5	4,130
print_token2	399	10	4,115
replace	512	31	5,542
schedule	292	9	2,650
schedule2	301	9	2,710
tcas	141	36	1,608
tot_info	440	19	1,051
space	6,218	35	13,585
NanoXML v1	3,497	6	214
NanoXML v2	4,007	7	214
NanoXML v3	4,608	9	216
NanoXML v5	4,782	8	216
XML security v1	21,613	6	92
XML security v2	22,318	6	94
XML security v3	19,895	4	84
Rhino	49k	11	20-152
Lucene	88k	9	1,072-1,154
Ant	264k	10	1,024-1,555

Zimmermann [8]. As for Lucene and Ant, we obtain their bugs and test suites from their bug tracking and version control systems following the procedure described by Dallmeier and Zimmermann [8]. In particular, we consider Lucene bugs that were reported for versions 2.0 to 3.1, and Ant bugs that were reported for versions 1.5.1 to 1.9.2. Kawrykow and Robillard observed that not all changes made to fix a bug are essential; many of them are cosmetic changes or simple refactorings that do not change the behavior of a program [19]. Thus, to find the root cause of a real bug (i.e., the buggy program blocks), we need to manually inspect the code that are changed to fix the bug. To help us in the manual inspection, we only include Rhino, Lucene, and Ant bugs whose fixes only change at most five lines of code. Among these bugs, we also only choose bugs that have at least one failed test case covering the faulty lines.

5.2 Evaluation Criteria

We evaluate the effectiveness of a fault localization technique (either using data fusion or not) in localizing faults based on the following commonly used metrics:

1) Percentage of Code Inspected: For each bug, we measure the percentage of program blocks that a developer needs to inspect to locate all the faulty program elements. This metric depends on the rank of the faulty program elements in the list. In this paper, we report the average percentage of program blocks inspected over all the bugs.

2) Proportion of Bugs Localized: We compute the proportion of bugs that can be localized when developers inspect up to a certain percentage of program blocks.

3) Absolute Amount of Code Inspected: As studied by Panin and Orso [29], developers may only inspect a certain number of most suspicious program elements recommended by an automatic debugging tool. Thus, we also compute the number of bugs that can be localized when developers inspect up to E program blocks. In this study, we set E to 10. We assume that 10 is still a reasonable number of program elements that developer would inspect.

5.3 Experiment Results

We evaluate the effectiveness of various variants of our approach to localize faults. We denote a variant of *Fusion Localizer* as $F_Z^{X,Y}$ where X specifies a score normalization method (i.e., Zero-One or Reciprocal), Y specifies a technique selection method (i.e., Overlap or Bias), and Z specifies a data fusion method such as CombANZ, CorrA_Top10%, CorrA_Top50%, CorrB_Top10%, CorrB_Top50%, CombSUM, CombMNZ, or Borda count.

Their effectiveness are compared with the effectiveness of the well-known spectrum-based fault localization techniques, namely Tarantula [18] and Ochiai [3]. We also compare with the theoretically best spectrum-based fault localization techniques [44] namely Naish1, Naish2, Binary, Wong1, and Russel & Rao, as well as the theoretically best genetic programming (GP) based fault localization techniques namely GP02, GP03, GP13, and GP19 [45].

Table 4: Average Percentage of Code Inspected to Localize All Bugs.

Technique	Average	Technique	Average
$F_{CombANZ}^{Zero-One,Overlap}$	21.36%	Naish2	24.63%
$F_{CombANZ}^{Zero-One,Bias}$	21.39%	GP13	24.78%
$F_{CombSUM}^{Zero-One,Bias}$	22.94%	Ochiai	25.29%
$F_{CorrB\_Top50\%}^{Zero-One,Overlap}$	23.11%	GP03	25.82%
$F_{CombSUM}^{Zero-One,Overlap}$	23.15%	Tarantula	26.77%
$F_{CorrB\_Top10\%}^{Zero-One,Overlap}$	23.23%	GP19	31.60%
$F_{CombMNZ}^{Zero-One,Overlap}$	23.31%	Naish1	34.40%
$F_{CorrB\_Top10\%}^{Zero-One,Bias}$	23.33%	GP02	39.48%
$F_{CorrB\_Top50\%}^{Zero-One,Bias}$	23.38%	Russel&Rao	42.48%
$F_{CorrA\_Top10\%}^{Zero-One,Overlap}$	23.56%	Binary	52.04%
$F_{CombMNZ}^{Zero-One,Bias}$	23.78%	Wong1	86.26%
$F_{CorrA\_Top10\%}^{Zero-One,Bias}$	23.78%		

5.3.1 Percentage of Code Inspected

In terms of average percentage of code inspected to localize all bugs, twelve variants of *Fusion Localizer* perform better than the state-of-the-art fault localization techniques. These variants fuse the techniques using CombANZ, CorrA_Top10%, CorrA_Top50%, CorrB_Top10%, CorrB_Top50%, CombSUM, and CombMNZ by first normalizing the scores using Zero-one normalization and selecting the techniques using either the bias-based or overlap-based method. They could achieve smaller average percentage of code inspected (i.e., 21.36% to 23.78%) as compared to the best performing state-of-the-art fault localization technique (i.e., Naish2). Among the state-of-the-art fault localization techniques, Naish2 achieves the smallest average percentage of code inspected (i.e., 24.63%), followed by GP13 (i.e., 24.78%) and Ochiai (i.e., 25.29%). See Table 4 for the details of average percentage of code inspected for the above variants of *Fusion Localizer* and the state-of-the-art fault localization techniques.

To investigate whether the differences in the average percentage of code inspected between our twelve variants and the best performing state-of-the-art fault localization techniques (i.e., Naish2, GP13, and Ochiai) are significant or not, we perform a statistical significance test namely Wilcoxon signed rank test [41] at 5% significance level. This statistical test does not assume that the data

should follow normal distribution. Based on the significance tests, $F_{CombANZ}^{Zero-One,Bias}$ and $F_{CombANZ}^{Zero-One,Overlap}$ significantly outperform Naish2 with p-values equal to 0.0135 and 0.02284, respectively. They also significantly outperform GP13 and Ochiai with p-values < 0.05. The other ten variants could only significantly outperform Ochiai with p-values <0.05. Table 5 lists the p-values when we compare each of the best performing *Fusion Localizer* variants with each of the best performing state-of-the-art fault localization techniques using Wilcoxon signed rank test.

Table 5: Statistical Significance Test Results

Fusion Localizer Variant	p-value over Ochiai	p-value over GP13	p-value over Naish2
$F_{CombANZ}^{Zero-One,Overlap}$	<0.0001	0.0182	0.0228
$F_{CombANZ}^{Zero-One,Bias}$	<0.0001	0.0104	0.0135
$F_{CombSUM}^{Zero-One,Bias}$	<0.0001	0.2464	0.2736
$F_{CorrB\_Top50\%}^{Zero-One,Overlap}$	<0.0001	0.2966	0.2985
$F_{CombSUM}^{Zero-One,Overlap}$	<0.0001	0.5985	0.6189
$F_{CorrB\_Top10\%}^{Zero-One,Overlap}$	<0.0001	0.5495	0.5642
$F_{CombMNZ}^{Zero-One,Overlap}$	<0.0001	0.6354	0.7402
$F_{CorrB\_Top10\%}^{Zero-One,Bias}$	<0.0001	0.2047	0.2464
$F_{CorrB\_Top50\%}^{Zero-One,Bias}$	<0.0001	0.2981	0.2948
$F_{CorrA\_Top10\%}^{Zero-One,Overlap}$	<0.0001	0.4842	0.5231
$F_{CombMNZ}^{Zero-One,Bias}$	0.0032	0.7860	0.8588
$F_{CorrA\_Top10\%}^{Zero-One,Bias}$	<0.0001	0.3065	0.3065

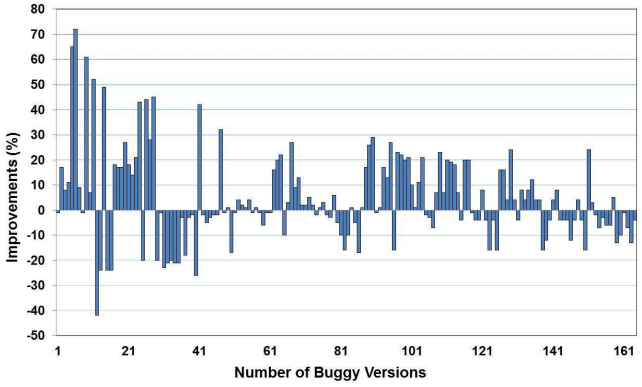

Figure 3: Improvement of $F_{CombANZ}^{Zero-One,Bias}$ over Naish2 in Terms of Percentage of Code Inspected

Furthermore, we plot the improvements of the best performing variant of *Fusion Localizer*, namely $F_{CombANZ}^{Zero-One,Bias}$ over the three best performing state-of-the-art fault localization techniques namely Naish2, GP13, and Ochiai for each of the buggy versions in Figures 3, 4, and 5, respectively. We plot the improvements made by $F_{CombANZ}^{Zero-One,Bias}$ because it achieves the most significant improvement over Naish2 (i.e., the lowest p-value over Naish2). The improvement is based on the difference in percentage of code inspected. The graphs only show the buggy versions where the improvements are either positive (our technique performs better) or negative (our technique performs worse). There are 87 buggy versions in which $F_{CombANZ}^{Zero-One,Bias}$ improves Naish2 and GP13. As compared to Ochiai, it has better performance for 91 buggy versions. In addition, there are 66, 65, and 88 buggy versions in which $F_{CombANZ}^{Zero-One,Bias}$ performs the same as Naish2, GP13, and Ochiai, respectively. From the figures, it is clear that our best variant outperforms the three best performing state-of-the-art fault localization techniques for most of the buggy versions.

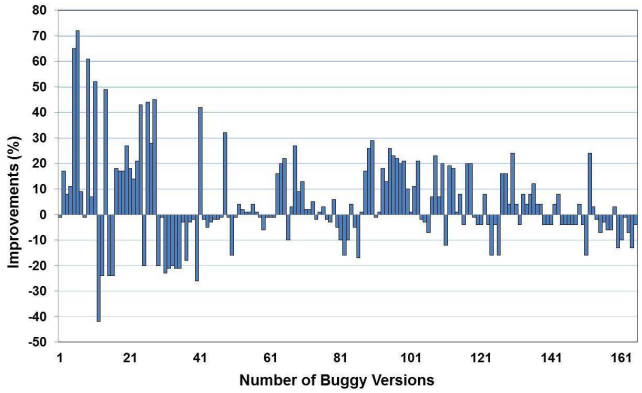

Figure 4: Improvement of $F_{CombANZ}^{Zero-One,Bias}$ over GP13 in Terms of Percentage of Code Inspected

5.3.2 *Proportion of Bugs Localized*

We also evaluate the number of buggy versions in which developers could find the faulty program elements by inspecting a certain percentage of program elements. Table 6 shows the proportion of bug localized when developers only inspect the top 10% of the most suspicious program elements produced by the state-of-the-art fault localization techniques and our twelve variants that have better average percentage of code inspected than the best performing state-of-the-art fault localization techniques.

When 10% of the most suspicious program elements are inspected, $F_{CombANZ}^{Zero-One,Overlap}$ and $F_{correlation-based}^{Zero-One,Bias}$ can localize more bugs as compared to the other variants of *Fusion Localizer* and the state-of-the-art fault localization techniques. They can localize 46.52% to 46.96% of the bugs, while the best performing state-of-the-art fault localization technique (i.e., Ochiai) can localize 42.17% of the bugs. Naish2 and GP13 can localize only 36.96% of the bugs.

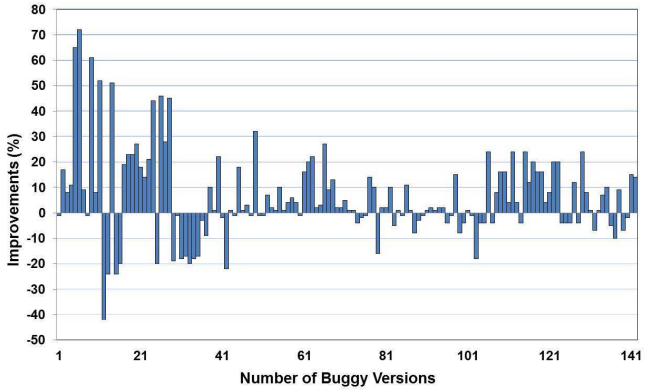

Figure 5: Improvement of $F_{CombANZ}^{Zero-One,Bias}$ over Ochiai in Terms of Percentage of Code Inspected

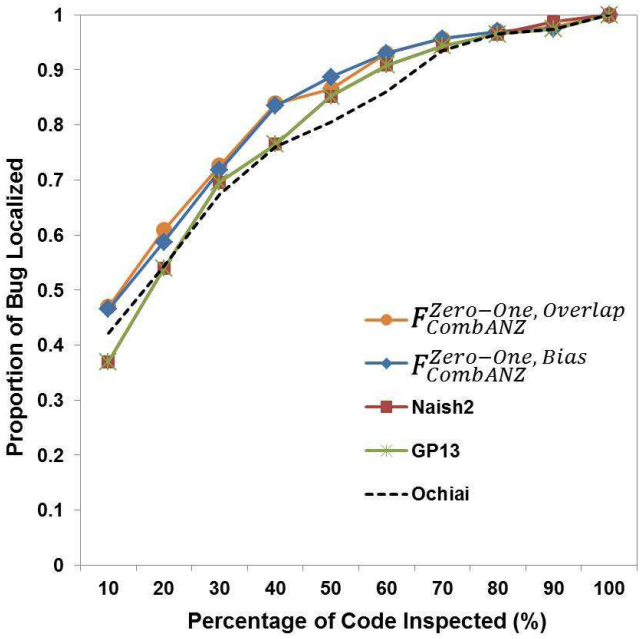

Figure 6: Comparing $F^{Zero-One,Bias}_{CombANZ}$, $F^{Zero-One,Overlap}_{CombANZ}$ with Naish2, GP13, and Ochiai

Thus, these two best variants could improve the best performing state-of-the-art fault localization techniques by 11.26% to 26.95%.

In addition, eight other variants of *Fusion Localizer* can localize more bugs than the best performing state-of-the-art fault localization technique (i.e., Ochiai) when 10% of program elements are inspected. They can localize 42.17% to 43.48% of the bugs. Thus, our best variants can improve number of bugs that can be localized when developers only inspect a small percentage of code.

Table 6: Proportion of Bug Localized When Only 10% of Blocks are Inspected.

Technique	%Bug	Technique	%Bug
$F^{Zero-One,Overlap}_{CombANZ}$	46.96%	$F^{Zero-One,Bias}_{CombMNZ}$	38.70%
$F^{Zero-One,Bias}_{CombANZ}$	46.52%	GP03	38.70%
$F^{Zero-One,Bias}_{CombSUM}$	43.48%	GP02	37.83%
$F^{Zero-One,Overlap}_{CombSUM}$	43.48%	Tarantula	37.39%
$F^{Zero-One,Bias}_{CorrB\_Top10\%}$	43.48%	Naish2	36.96%
$F^{Zero-One,Overlap}_{CorrB\_Top10\%}$	43.48%	GP13	36.96%
$F^{Zero-One,Overlap}_{CorrA\_Top10\%}$	43.48%	Naish1	36.09%
$F^{Zero-One,Bias}_{CorrA\_Top10\%}$	43.04%	GP19	29.13%
$F^{Zero-One,Bias}_{CorrB\_Top50\%}$	43.04%	Russel&Rao	5.65%
$F^{Zero-One,Overlap}_{CorrB\_Top50\%}$	42.61%	Binary	4.78%
Ochiai	42.17%	Wong1	3.04%
$F^{Zero-One,Overlap}_{CombMNZ}$	41.30%		

Furthermore, we plot the proportion of bug localized using our two best variants (i.e., $F^{Zero-One,Overlap}_{CombANZ}$ and $F^{Zero-One,Bias}_{CombANZ}$), Naish2, GP13, Ochiai, and Tarantula when different percentage of program elements are inspected, as shown in Figures 6. Based on the figure, our two best variants can localize more bugs when different percentages of program elements are inspected as compared to the best performing state-of-the-art fault localization techniques.

Table 7: Number of Bugs Localized When Only up to 10 Most Suspicious Program Elements are Inspected (i.e., Hit@10)

Technique	Hit@10	Technique	Hit@10
$F^{Zero-One,Bias}_{CombANZ}$	91	Ochiai	74
$F^{Zero-One,Overlap}_{CombANZ}$	87	Naish1	73
$F^{Zero-One,Overlap}_{CombSUM}$	87	Naish2	73
$F^{Zero-One,Bias}_{CorrA\_Top10\%}$	86	GP13	72
$F^{Zero-One,Overlap}_{CorrA\_Top10\%}$	86	GP03	71
$F^{Zero-One,Bias}_{CorrB\_Top10\%}$	85	GP02	64
$F^{Zero-One,Overlap}_{CorrB\_Top10\%}$	85	Tarantula	56
$F^{Zero-One,Bias}_{CorrB\_Top50\%}$	85	GP19	51
$F^{Zero-One,Overlap}_{CorrB\_Top50\%}$	84	Russel&Rao	3
$F^{Zero-One,Bias}_{CombSUM}$	84	Binary	3
$F^{Zero-One,Overlap}_{CombMNZ}$	84	Wong1	0
$F^{Zero-One,Bias}_{CombMNZ}$	78		

5.3.3 *Absolute Amount of Code Inspected*

We also investigate the number of bugs that can be localized when only a small number of program elements (i.e., 10 program blocks) are inspected. The results are shown in Table 7.

The table shows that our twelve variants can localize more bugs than the state-of-the-art fault localization techniques. They can localize 78 to 91 bugs, while the best performing state-of-the-art fault localization technique (i.e., Ochiai) can only localize 74 bugs. Amongst our variants, $F^{Zero-One,Bias}_{CombANZ}$ can localize the largest number of bugs as compared to the other techniques. Its relative improvement over Ochiai, Naish2, and GP13 are 23%, 25%, and 26%, respectively. These results also show that our best variants can substantially improve the number of bug localized when developers only inspect a small number of program elements.

5.4 Discussion

In this section, we discuss the effect of score normalization, technique selection, and varying the number of techniques to be fused together on the performance of *Fusion Localizer*. At the end of this section, we describe some threats to validity.

Effect of Score Normalization. To investigate the effect of the score normalization step, we disable this step and evaluate the effectiveness of the resultant solution. We find that without score normalization, the performance of the best performing variant of *Fusion Localizer* is not as good as the best performing variant that normalizes the scores.

Without normalization, the average percentage of program block inspected to localize all bugs achieved by the best variant is 30.28% which is larger than the result of the best variant that normalizes the scores (i.e., 21.36%). When developers only inspect up to 10% of program blocks, the best variant that does not normalize the scores can only localize 30% of the bugs which is smaller than the percentage of bugs that are successfully localized by the best variant that normalizes the scores (i.e., 46.96%). Also, when inspecting up to 10 program blocks, the number of bugs localized by the best variant that does not normalize the scores is not even half of those

achieved by the best variant that normalizes the scores (38 bugs vs. 91 bugs). Therefore, normalization is an important step to improve the performance of *Fusion Localizer*.

Effect of Technique Selection. To investigate the effect of the technique selection step, we disable this step and evaluate the effectiveness of the resultant solution. When we fuse all 42 fault localization techniques, the performance of the best variant is worse than the best variant that employs technique selection. When the technique selection step is disabled, on average, the best variant can localize all bugs when 22.27% of code are inspected, which is not as good as the result achieved by the best variant that employs technique selection (i.e., 21.36%). When developers only inspect up to 10% of program blocks, the best variant that does not use technique selection localizes a smaller percentage of bugs than the percentage localized by the best variant that uses technique selection (i.e., 46.52% vs. 46.96%). Similarly, when developers only inspect up to 10 program blocks, the best variant that does not use technique selection localizes a smaller number of bugs than the number localized by the best variant that employs technique selection (i.e., 87 vs. 91). Therefore, applying technique selection can improve the performance of *Fusion Localizer*.

Effect of Number of Techniques to be Fused. To evaluate the effect of the number of techniques to be fused by our *Fusion Localizer*, we use one of our best variants (i.e., $F^{Zero-One,Bias}_{CombANZ}$) and set the number of techniques to be selected to top 25%, top 50%, top 75%, and all of the techniques. Let us refer to the 4 sub-variants of $F^{Zero-One,Bias}_{CombANZ}$ as V25, V50, V75, and V100. By default *Fusion Localizer* selects top 50% of the techniques. We find that the average percentage of code inspected to locate all bugs are 27.13%, 21.36%, 21.88%, and 22.27% for V25, V50, V75, and V100, respectively. When developers only inspect the top 10% of the code, the percentage of bugs localized by V25, V50, V75, and V100 are 30%, 46.96%, 46.52%, and 46.52%, respectively. When developers only inspect the top 10 program blocks, the number of bugs localized by V25, V50, V75, and V100 are 57, 91, 87, and 87, respectively. The results show that selecting the top 50% of the techniques (the default option) is the best setting.

Threats to Validity. Threats to internal validity relates to errors in our experiments. We have checked our implementation but there could be bugs that we do not notice. Threats to external validity relates to the generalizability of our findings. We have analyzed 230 bugs from 13 programs written in C and Java. The programs vary from small to large programs. The bugs vary from synthetic to real bugs. In the future, we plan to reduce this threat to external validity further by investigating more bugs from more systems written in various programming languages.

Threats to construct validity relates to the suitability of our evaluation metrics. We have used common metrics used to analyze past fault localization studies [3, 17, 18, 26, 27]. We have also used another metric (i.e., Hit@10) to address the concern raised by Parnin and Orso [29]. Hit@10 only considers the performance of a fault localization tool when a small number of program elements (in our case: program blocks) are inspected. The measure was previously used by information retrieval (IR) based bug localization studies that find buggy program files given a textual bug report [31, 35, 38, 40, 43, 51]. Admittedly, there is no *large scale* study that shows positive correlation (or its absence) between improving Hit@10 (or other rank-based metrics) and time and effort saved when developers use a fault localization technique to debug various kinds of bugs. Still, improving rank-based metrics is a step towards building a fault localization technique that can pinpoint the location of bugs in the top position most of the time. Such a technique will be highly effective since developers can *trust* its output. The location pinpointed by such tool can be a good starting point for developers to reason on the root cause behind a set of failures and how to fix the bug. This location can also be used as input to other studies, for example, automated bug fixing [13].

6. CONCLUSION AND FUTURE WORK

In this paper, we propose an approach named *Fusion Localizer* to fuse a number of spectrum-based fault localization techniques. Our propose approach consists of three steps: score normalization, technique selection, and data fusion. We investigate two score normalization methods, two technique selection methods, and five data fusion methods resulting in twenty variants of *Fusion Localizer*. *Fusion Localizer* does not require any training data, i.e., execution traces of past relevant program failures, to select the set of techniques to be fused. This allows our approach to be used for new programs or programs where program spectra of past relevant bugs are unavailable. Furthermore, our approach is bug specific in which the set of techniques to be fused is adaptively selected for each buggy program based on its spectra.

We evaluate our approach using a common benchmark dataset and a dataset that contains real bugs from three medium to large programs. Our evaluation shows that the best performing variants of *Fusion Localizer* (i.e., $F^{Zero-One,Bias}_{CombANZ}$ and $F^{Zero-One,Overlap}_{CombANZ}$) statistically significantly outperform the state-of-the-art spectrum-based fault localization techniques (i.e., Ochiai, theoretically best spectrum-based fault localization techniques, and theoretically best genetic programming (GP) based fault localization techniques). Our best variants require smaller average percentage of code inspected to locate faulty elements as compared to the best performing state-of-the-art fault localization techniques (i.e., 21% vs. 24%). When developers only inspect 10% of the most suspicious program elements, these best variants could improve the best performing state-of-the-art fault localization techniques (i.e., Ochiai, Naish2, and GP13) by 11% to 26%. Furthermore when developers only inspect the top 10 most suspicious program blocks, these best variants could improve the best performing state-of-the-art fault localization techniques by 23% to 26%. In addition, there are many other variants of *Fusion Localizer* that can outperform the state-of-the-art fault localization techniques (albeit not statistically significantly). These variants use CorrA_Top10%, CorrB_Top10%, CorrB_Top50%, CombSUM, and CombMNZ to fuse the selected techniques, by first normalizing the scores using Zero-One normalization and by selecting the techniques using the bias-based or overlap-based selection methods.

As a future work, to improve the effectiveness of our proposed approach further, we plan to create new technique selection and fusion methods that are specifically designed for fault localization. Also, rather than only returning a list of most suspicious program elements, we plan to extend studies on bug signature mining, e.g., [5, 15, 25], and return additional contextual information that can help developers debug. Additionally, we plan to perform a user study to evaluate the effectiveness of our technique in saving developer time and effort when localizing real bugs from various subject programs written in various programming languages.

Acknowledgement

We would like to thank Tien-Duy B. Le for his help in collecting the real bugs and test suites from Lucene and Ant bug tracking and version control systems following the approach by Dallmeier and Zimmermann [8].

7. REFERENCES

[1] Ant. http://svn.apache.org/.

[2] Lucene-jira. https://issues.apache.org/jira/browse/LUCENE.

[3] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evalution on spectrum-based fault localization. *The Journal of System and Software*, 82:1780–1792, 2009.

[4] J. A. Aslam and M. Montague. Models for metasearch. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'01)*, pages 276–284. ACM, 2001.

[5] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *ISSTA*, pages 141–152, 2009.

[6] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *IEEE 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society, May 2009.

[7] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 342–351, St. Louis, Missouri, May. 2005.

[8] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE'07)*, pages 433–436, 2007.

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

[10] E. A. Fox, M. P. Koushik, J. Shaw, R. Modlin, D. Rao, et al. Combining evidence from multiple searches. In *The first text retrieval conference (TREC-1)*, pages 319–328. US Department of Commerce, National Institute of Standards and Technology, 1993.

[11] E. A. Fox and J. A. Shaw. Combination of multiple searches. *NIST SPECIAL PUBLICATION SP*, pages 243–243, 1994.

[12] L. Gong, D. Lo, L. Jiang, and H. Zhang. Interactive fault localization leveraging simple user feedback. In *ICSM*, pages 67–76, 2012.

[13] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[14] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *In Proc. of the 20th IEEE/ACM Int. Conf. on Automated Engineering, ASE*, pages 263–272, 2005.

[15] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *ASE*, pages 439–442, 2008.

[16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 191–200, 1994.

[17] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *In Proc. of the 22th Int. Conf. on Software Engineering, ICSE*, pages 467–477, 2002.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault localization technique. In *In Proc. of the 20th IEEE/ACM Int. Conf. on Automated Engineering, ASE*, pages 273–282, 2005.

[19] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 351–360, 2011.

[20] T.-D. B. Le and D. Lo. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *ICSM*, pages 310–319, 2013.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation, PLDI*, pages 141–154, San Diego, CA, 2003.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN 2005 Int. Conf. Programming Language Design and Implementation, PLDI*, 2005.

[23] D. Lillis, F. Toolan, R. Collier, and J. Dunnion. Probfuse: a probabilistic approach to data fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 139–146. ACM, 2006.

[24] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: Statistical model-based bug localization. In *Proc. of Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[25] D. Lo, H. Cheng, and X. Wang. Bug signature minimization and fusion. In *HASE*, pages 340–347, 2011.

[26] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *In Proc. of the 26th IEEE Int. Conf. on Software Maintenance, ICSM*, pages 1–10, 2010.

[27] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software Evolution and Process*, 2013.

[28] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[29] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis(ISSTA'11)*, pages 199–209, 2011.

[30] R. R. Nuray and F. Can. Automatic ranking of information retrieval systems using data fusion. *Information Processing & Management*, 42(3):595–614, 2006.

[31] S. Rao and A. C. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.

[32] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *In Proc. of the 18th IEEE/ACM Int. Conf. on Automated Engineering, ASE*, pages 141–154, 2003.

[33] S. Roychowdhury and S. Khurshid. A family of generalized entropies and its application to software fault localization. In *IEEE Conf. of Intelligent Systems*, pages 368–373, 2012.

[34] S. Roychowdhury and S. Khurshid. Localization of faults in software programs using bernoulli divergences. In *Information Theory and its Applications (ISITA), 2012 International Symposium on*, pages 586–590, 2012.

[35] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[36] R. Santelices, J. Jones, Y. Yu, and M. Harrold. Lightweight fault-localization using multiple coverage types. In *In Proc. of the 29th Int. Conf. on Software Engineering, ICSE*, 2009.

[37] Siemens, M. Harrold, and G. Rothermel. *Aristotle Analysis System – Siemens Programs, HR Variants*. http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[38] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *ICPC*, 2014.

[39] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau. Search-based fault localization. In *In Proc. of the 26th IEEE/ACM Int. Conf. on Automated Engineering, ASE*, pages 556–559, 2011.

[40] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *ICSME*, 2014.

[41] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin available: http://www.jstor.org/stable/3001968*, 1:80–83, 1945.

[42] S. Wu. *Data Fusion in Information Retrieval*, volume 13. Springer, Heidelberg, 2012.

[43] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang. Cross-language bug localization. In *ICPC*, pages 275–278, 2014.

[44] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[45] X. Xie, F. Kuo, T. Y. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *Search Based Software Engineering*, pages 224–238. Springer, 2013.

[46] R. B. Yates and B. R. Neto. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[47] S. Yoo. Evolving human competitive spectra-based fault localisation techniques. In *Search Based Software Engineering*, pages 244–258. Springer, 2012.

[48] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2 edition, 2009.

[49] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 28:183–200, 2002.

[50] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *In Proc. of the 26th Int. Conf. on Software Engineering, ICSE*, 2006.

[51] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.