Towards More Accurate Fault Localization: An Approach Based on Feature Selection Using Branching Execution Probability

Ang Li[1] Yan Lei[2] Xiaoguang Mao[1]
[1]National University of Defense Technology, China
[2]Logistical Engineering University of PLA, China
angli.cs@outlook.com, yanlei.cs@outlook.com, xgmao@nudt.edu.cn

***Abstract*—The current fault localization techniques for debugging basically depend on the binary execution information which indicates each program statement being executed or not executed by a particular test case. However, this simple information may lose some essential clues such as the branching execution information for fault localization, and therefore restricts localization effectiveness. To alleviate this problem, this paper proposes a novel fault localization approach denoted as FLBF which incorporates the branching execution information in the manner of feature selection. This approach firstly uses branching execution probability to model the behavior of each statement as a feature, then adopts one of the most widely used feature selection method called Fisher score to calculate the relevance between each statement's feature and the failures, and finally outputs the suspicious statements potentially responsible for the failures. The scenario used to demonstrate the utility of FLBF is composed of two standard benchmarks and three real-life UNIX utility programs. The experimental results show that input with branching execution information can improve the performance of current fault localization techniques and FLBF performs more stably and efficiently than other six typical fault localization techniques.**

***Index Terms*—fault localization, branching execution probability, feature selection.**

I. Introduction

Being a tedious and difficult task in software development and maintenance, debugging usually requires developers to consume a significant amount of time and resources in pinpointing the location of a bug and understanding its cause of a failure [1]. In order to improve debugging performance, researchers has devoted much effort to developing *automated fault localization* techniques such as [2]–[13].

Many of these techniques usually use the binary execution information which refers to the information of each program statement whether being executed or not executed by a particular test case. Based on the binary execution information and test results, these localization techniques adopt an evaluation formula to evaluate the suspiciousness of each statement being faulty and output a ranked list of all statements in descending order of suspiciousness. In summary, the basic intuition of these techniques is that if a statement is executed by a failing test case, its suspiciousness of being faulty will increase; on the contrary, if a statement is executed by a passing test case, its suspiciousness will decrease.

Nevertheless, we can observe that the binary execution information only shows whether a statement is executed or not executed by a test case, and thus can miss some useful clues such as the branching information for fault localization. It is evident that branching structure is one of the most common structures in program design and different condition satisfaction leads to the executions of different branches [14]. In this respect, if we ignore these essential information from the program, it may potentially restrict the effectiveness and accuracy of fault localization. To elucidate this point, here we give a simple example below.

Suppose that a faulty statement S_f and a non-faulty statement S_n belong to two different branching modules. Next, we assume that the two statements are executed or not executed by the same test cases. In this way, the two statements should have the same information of executed or not executed by the test cases, and thus the current fault localization techniques will assign the same suspiciousness of being faulty to the two statements. Since the two statements belong to two different branching modules, the branching probability of S_f should be different from that of S_n. Meanwhile, the execution of the two statements turn out to be different. Therefore, the two statements' suspiciousness of being faulty are supposed to be unequal and the current fault localization techniques fail to take this apparent omission into account. In other words, it means that by incorporating branching probability into fault localization, we can distinguish more statements' behaviors and enrich the resource data which fault localization relies on, and this may potentially improve the effectiveness and the accuracy of fault localization.

Therefore, this paper tries to justify the importance of branching information and incorporate it into the process of fault localization. Since the calculation of theoretical branching probability is complicated and even infeasible in practice, we introduce a concept of branching execution probability, that is, the ratio of the execution times of a statement located in a branching module to that of the entire corresponding branching module. For example, suppose a branching module contains two branches: the true branch and the false branch. The whole branching module is executed 10 times by a test case, and a statement located in the false branch is executed

6 times during the whole 10 times executions. Therefore the branching execution probability of this statement turns out to be 0.6 (it is easy to calculate as 6/10=0.6), In contrast to theoretically defined branching probability which tends to be complex and difficult to be applied in practical situations, branching execution probability mentioned above is easy to calculate and obtain from the execution(s) of a test case or a set of test cases. Therefore, we can utilize the branching execution probability to describe each statement's behaviors and obtain more subtle and useful information compared with the traditional simple information of a statement being executed or not executed.

Feature selection [15], in machine learning and statistics, is a method selecting a subset of relevant features in the presence of a reference feature. Inspired by feature selection, we plan to distinguish the suspicious statements by leveraging feature selection to find the relevance between the feature of a statement and that of the test results. Specifically, we propose a Fault Localization approach using Branching execution probability in the manner of Feature selection, denoted as `FLBF`. `FLBF` first uses the branching execution probability to describe the behavior of each statement as a feature. Next, we define the binary information to represent the test results as a reference feature. Furthermore, `FLBF` adopts feature selection to calculate the relevance between each statement's feature and the reference feature. Finally, our approach distinguishes the suspicious statements in terms of scores evaluated by feature selection.

The contributions of our study are summarized as follows:

- We explore the potential of branching execution probability and propose an approach leveraging branching execution probability to capture more subtle and essential information for improving the efficiency and accuracy of fault localization.
- We improve representative fault localization techniques using branching execution probability and experiment results show evident enhancement in their performance.
- We successfully adopt the methodology of feature selection to incorporate branching execution probability into the process of fault localization and demonstrate the promising prospect of feature selection techniques in fault localization.
- We use common data sets and real-life UNIX utility programs to evaluate our approach and show its effectiveness in improving fault localization.

The remainder of this paper is organized as follows. Section II introduces necessary background information. Section III presents our approach. Section IV shows the experimental results and analysis, and Section V draws the conclusion.

II. Background

This section will introduce those fault localization techniques using binary execution information (that is the information of a statement being executed or not executed), and the methodology of feature selection.

$$\begin{array}{c c c}
 & \textit{M statements} & \textit{failing/passing} \\
\textit{N test cases} & \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1M} \\ x_{21} & x_{22} & \cdots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NM} \end{bmatrix} & \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{bmatrix} \begin{matrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{matrix} \\
 & \begin{matrix} s_1 & s_2 & \cdots & s_M \end{matrix} &
\end{array}$$

Fig. 1. Input to Fault Localization Using Binary Execution Information.

A. Fault Localization Using Binary Execution Information

Here we describe the principle of fault localization techniques using binary execution information [5]. Suppose that there is a program $\mathcal{P}$ with its statements $\mathcal{S} = \{s_1, s_2, ..., s_M\}$ running against the test suite $\mathcal{T} = \{t_1, t_2, ..., t_N\}$.

Fig. 1 illustrates the input of these techniques, which is a $N \times (M + 1)$ matrix. The element x_{ij} equals 1 if the statement s_j is executed by the test case t_i, and 0 otherwise. The result vector r at the rightmost column of the matrix denotes the test results. The element r_i is 1 if t_i is a failed test case, 0 otherwise.

To evaluate the suspiciousness of a statement, these techniques normally utilize the similarity between the statement vector and result vector of the matrix in Fig. 1. Four statistical variables are defined for each statement to conduct the similarity computation, as shown in Eq. 1.

$$\begin{aligned}
a_{00}(s_j) &= |\{i|x_{ij} = 0 \land r_i = 0\}| \\
a_{01}(s_j) &= |\{i|x_{ij} = 0 \land r_i = 1\}| \\
a_{10}(s_j) &= |\{i|x_{ij} = 1 \land r_i = 0\}| \\
a_{11}(s_j) &= |\{i|x_{ij} = 1 \land r_i = 1\}|
\end{aligned} \tag{1}$$

From Eq. 1, we can observe that $a_{00}(s_j)$ represents the number of *passing* test cases which does *not execute* the statement s_j; $a_{01}(s_j)$ denotes the number of *passing* test cases which *executes* the statement s_j; $a_{10}(s_j)$ stands for the number of *failing* test cases which does *not execute* s_j; $a_{11}(s_j)$ is the number of *failing* test case which *executes* the statement s_j.

In recent years, many different suspiciousness evaluation formulas sprouted up with the help of these four variables to evaluate the suspiciousness of a statement being faulty and generate a ranked list of suspiciousness with all statements in descending order. In order to conduct a comprehensive evaluation for our approach, we choose six typical fault localization techniques, among which three are typical human-deisnged techniques (Ochiai [5], Tarantula [4] and Jaccard [18]) and the other three are machine-learned ones (GP02 [19], GP03 [19] and GP19 [19]). Table I lists these six representative suspiciousness evaluation formulas with their specific definitions and shows their own means of calculating the suspiciousness value of the statement s_j.

B. Feature Selection

Feature selection (also known as variable selection or attribute selection) is a process of selecting a subset of relevant features (such as variables, predictors) for use in model construction [20], [21]. To be more precise, feature

TABLE I
SUSPICIOUSNESS EVALUATION FORMULAS.

Name	Formula
Ochiai	$\frac{a_{11}(s_j)}{\sqrt{(a_{11}(s_j)+a_{01}(s_j))*(a_{11}(s_j)+a_{10}(s_j))}}$
Tarantula	$\frac{(\frac{a_{11}(s_j)}{a_{11}(s_j)+a_{01}(s_j)})}{(\frac{a_{11}(s_j)}{a_{11}(s_j)+a_{01}(s_j)})+(\frac{a_{10}(s_j)}{a_{10}(s_j)+a_{00}(s_j)})}$
Jaccard	$\frac{a_{11}(s_j)}{a_{11}(s_j)+a_{01}(s_j)+a_{10}(s_j)}$
GP02	$2(a_{11}(s_j)+\sqrt{a_{00}(s_j)})+\sqrt{a_{10}(s_j)}$
GP03	$\sqrt{\lvert a_{11}(s_j)^2-\sqrt{a_{10}(s_j)}\rvert}$
GP19	$a_{11}(s_j)\sqrt{\lvert a_{11}(s_j)-a_{01}(s_j)+a_{00}(s_j)-a_{10}(s_j)\rvert}$

selection methods are usually implemented to identify and remove unneeded, irrelevant and redundant attributes from data that do not contribute to the accuracy of a predictive model or may in fact decrease the accuracy of the model [22]. Therefore, those methods have a wide range of applications in reducing the amount of data to deal with and the effect of the noise produced in the process, enhancing the performance of the whole system. The objective of feature selection has three different levels: enhancing the prediction success rate of the predictors, presenting more efficient and cost-effective predictors, and producing a deeper understanding of the underlying process which produced the data. Generally, there are three families of feature selection methods: filter methods, wrapper methods and embedded methods, among which the filter-based methods rank the features as a pre-processing step prior to the learning algorithm, and select those features with high ranking scores [23]. In our study, we focus on one of the most widely used filter-based supervised methods for feature selection called Fisher score.

In recent studies, Fisher score(or Fisher kernel) is increasingly utilized as an effective feature extractor for the problems such as dimensionality reduction and classification [24], [25]. Basically, the Fisher score refers to a vector of parameter derivatives of loglikelihood in a complicated model [25]. The main idea of Fisher score aims at finding a subset of features so that in the data space extended by the selected features, the distances among the data points in the same class are as small as possible, whereas the distances among the data points in different classes are as large as possible [24]. It selects the feature of each element independently according to the scores they obtain under the Fisher criterion and produce a suboptimal subset of features. This can help to transform the input vectors of variable length to fixed-length vectors effectively. More specifically, there are m selected features, and the input data matrix $\mathbf{X} \in \mathbb{R}^{d \times n}$ declines to $\mathbf{Z} \in \mathbb{R}^{m \times n}$. Then, we compute the Fisher score as follows:

$$F(\mathbf{Z}) = tr\{(\tilde{\mathbf{S}}_b)(\tilde{\mathbf{S}}_t + \gamma \mathbf{I})^{-1}\} \tag{2}$$

where γ represents a positive regularization parameter, $\tilde{\mathbf{S}}_b$ is called between-class scatter matrix, and $\tilde{\mathbf{S}}_t$ indicates total scatter matrix, which are defined as

$$\begin{aligned} \tilde{\mathbf{S}}_b &= \sum_{k=1}^{c} n_k(\tilde{\mu}_k - \tilde{\mu})(\tilde{\mu}_k - \tilde{\mu})^{\mathrm{T}} \\ \tilde{\mathbf{S}}_t &= \sum_{i=1}^{n} (z_i - \tilde{\mu})(z_i - \tilde{\mu})^{\mathrm{T}} \end{aligned} \tag{3}$$

where $\tilde{\mu}_k$ and n_k represent the mean vector and the size of the k-th class in the reduced data space. Then we select the top-m ranked features with the highest scores after calculating the Fisher score for each feature. The higher score one feature obtains, the more relevant it tends to be with the selected features. In terms of fault localization, we are enlightened by the idea of Fisher score and plan to explore the potential of it by implementing our method in the following pages (See Section III).

Feature selection has shown its wide application in fields like machine learning and pattern recognition [26], but for the best of our knowledge no one has implemented it in fault localization. In the following part we will explain its enormous potential for fault localization problems.

III. FAULT LOCALIZATION USING BRANCHING EXECUTION PROBABILITY WITH FEATURE SELECTION

In this section, we present the algorithm of `FLBF` (**Fault Localization** using **Branching** execution probability with **Feature** selection), showing the methodology of `FLBF` using branching execution probability in the manner of feature section. The details of `FLBF` are described with three main steps as follows:

Step 1: Define new input matrix using branching execution probability. As shown in Fig. 1, the input of fault localization techniques using binary execution information is a $N \times (M+1)$ matrix. In their matrix, an element x_{ij} equals 1 or 0 according to whether the statement s_j is executed by test case t_i or not. However, it fails to distinguish more subtle statements' behaviors in the program execution, *e.g.*, the branching execution probability of the statements in the branching module. In order to capture more useful behaviors, we define new input matrix by using branching execution probability in this step.

The new input matrix is also a $N \times (M+1)$ matrix. In contrast to the matrix defined in Figure 1, the new matrix presents a different consideration for those statements in the branching modules, that is, the execution information of those statements in the branching modules should contain the branching execution information rather than just the simple information of whether executed or not executed by the test cases. Therefore, in the new input matrix, there are three possible situations for a certain element x'_{ij}:

- If the statement s_j which exists outside any branching module is covered by the execution of test case t_i, then the element x'_{ij} equals 1.
- If the statement s_j which exists outside any branching module is not covered by the execution of test case t_i, then the element x'_{ij} equals 0.

- If the statement s_j belongs to a certain branching module in the program, then the element x'_{ij} should be a decimal between 0 and 1, which denotes the branching execution probability of the statement s_j during the execution of t_i.

By this way, we successfully extend the original input matrix to contain the essential information of branching execution in the program.

Step 2: Calculate the branching execution probability. This step aims at calculating the branching execution probability of the statements located in branching modules with the new input matrix.

To facilitate the understanding of the new input matrix, we demonstrate the calculation of branching execution probability along with the elements which are already defined in the above new input matrix. Suppose that statement s_j belongs to the module of a certain branching statement s_j^{if}, and is executed et_i times by a test case t_i. Then, we let $et_i^{if(j)}$ be the execution times of s_j^{if} in the test case t_i. Consequently, we can now define the branching execution probability of the statement s_j in the test case t_i, which is the value of x_{ij} in the new input matrix, as follows:

$$x_{ij} = \frac{et_i}{et_i^{if(j)}} \tag{4}$$

Furthermore, it is common to see nested branching structure in different programs, *e.g.* the program segment from `print_tokens` as shown in Fig. 2. To include the information of nested branching structure, we need to enrich our definition based on the formula 4. Since the statements of a nested branching module already owns an execution probability p' before the execution jumps into those statements, thus the branching execution probability of each statement in this nested branching module is supposed to be multiplied by p'. In this case, the branching execution probability of the statement s_j in t_i is defined as follows:

$$x_{ij} = \frac{et_i \times p'}{et_i^{if(j)}} \tag{5}$$

In order to identify the branching modules belonging to which branching statement, we search for key words (e.g., if, switch) which stand for branching cases in the compiling phase.

Step 3: Evaluate each statement's suspiciousness value using Fisher score. As described in Section II-B, feature selection can evaluate a feature's relevance with the accuracy of the model. Inspired by feature selection and especially by filter-based methods, we treat each statement's behavior as a feature, and evaluate each feature's contribution to the test results, as the metric to measure the suspiciousness of each statement being faulty. Specifically, the vector of each statement in the input matrix of Fig.3 is a type of expression showing each statement's behavior in the program executions. The rightmost vector is a binary expression denoting the test results of all test cases. Thus, we use the vector of each statement in the input matrix as a feature representing their behaviors, and the one of test results as a reference feature referring to the relevance of each statement's feature. Therefore, we adopt Fisher score to evaluate the suspiciousness value of each statement being faulty, that is, the relevance of each statement's feature with the reference feature. The algorithm of Fisher score using branching execution information is described as follows:

```
static int next_state(state,ch)
int state;
char ch;
{
  if(state < 0)
    return(state);
  if(base[state]+ch >= 0)
  {
   if(check[base[state]+ch] == state)
    return(next[base[state]+ch]);
   else
    return(next_state(default1[state],ch));
  }
  else
    return(next_state(default1[state],ch));
}
```

Fig. 2. A Program Segment from *print_tokens*.

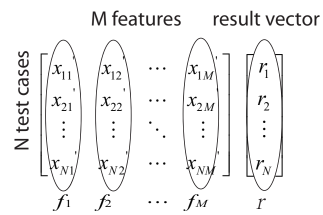

Fig. 3. New input matrix for Fisher score.

Firstly, we choose the result vector $r = \{r_1, r_2, ..., r_N\}$ as the reference feature. Next, for a statement s_j in $\mathcal{S} = \{s_1, s_2, ..., s_M\}$, there is a feature belongs to it, which is the vector $f_j = \{x'_{1j}, x'_{2j}, ..., x'_{Nj}\}$. After that, we apply Fisher score to calculate the relevance of each statement's feature f_j (where, $j \in \{1, ..., N\}$) based on the reference feature r. Finally, each statement obtains a fisher score, and a higher fisher score indicates a stronger correlation with the test results. Therefore, we obtain the fisher score to evaluate the suspiciousness value being faulty, and then rank all the statements in descending order according to their Fisher score value. A higher suspiciousness value indicates a higher probability of being the root cause of failures.

IV. Experimental Study

A. Experimental Setup

TABLE II
THE SUMMARY OF SUBJECT PROGRAMS.

Program	Versions	LoC	Test	Description
print_tokens (2 ver.)	15	570/726	4115/4130	Lexical analyzer
replace	27	564	5542	Pattern recognition
schedule (2 ver.)	16	374/412	2650/2710	Priority scheduler
tcas	29	173	1608	Altitude separation
tot_info	18	565	1052	Info. measure
space	35	6199	4333	ADL interpreter
flex	53	10459	567	Lexical analyzer
grep	29	14427	370	Pattern match
sed	29	14427	370	Stream editor

To evaluate our approach FLBF, the experiments chose the subject C programs widely used in most recent work as the testing benchmarks, namely `Siemens`, `space`, `flex`, `grep` and `sed`. The `Siemens` were originally developed at the Siemens Research Corporation and contain 7 programs in total as shown in Table II [8]. A number of faulty versions with single seeded faults are produced from these programs. The program `space` was first written by the European Space Agency and contains dozens of faulty versions with single real faults. Besides the two standard benchmarks, we use three real-life UNIX utility programs with real and seeded faults to strengthen the effect of our experiment. The three programs are `flex`, `grep` and `sed` respectively. Each of these programs has several sequential, previously-released versions, and each version contains dozens of single faults. All the subject programs of the study are obtained from the Software-artifact Infrastructure Repository (SIR[1]). In order to guarantee the universality and preciseness of data, the experiments select the *universe* test suite that includes all the test cases provided for each subject program.

Table II shows the information of subject programs and test suites that we use. The data include the programs (column "Program"), the number of faulty versions used by our experiment (column "Versions"), the lines of code (column "Loc"), the number of test cases in the *universe* test suite (column "Test"), as well as the functional description of the corresponding program (column "Description"). Because *print_tokens* and *print_tokens2* have similar structure and functionality, and each has only a few faulty versions, our experiments show their combined results to give meaningful statistics. Similarly, we also combine the results of *schedule* and *schedule2*.

Here we divide our experiments into two scenarios. In the first scenario, we generate input information of all the fault localization techniques in this experiment only using binary execution information, even for our approach `FLBF`. In the second scenario, we enrich the input information using branching execution probability. In this way, we can obtain experimental results of four situations: the six representative fault localization techniques using binary execution information, the six representative fault localization techniques using branching execution probability, Feature score using binary execution information and Feature score using branching execution information. It is also easy for us to check the influence of branching execution probability on the final localization results.

Our study implemented all the experiments on an Ubuntu 10.04 environment. We use the tool *gcov* to obtain the branching execution information of the statements in our benchmark programs.

B. Evaluation Metric

We use the absolute rank of faulty statements in the ranked list of all statements' suspiciousness value as a metric to evaluate the effectiveness of fault localization techniques, as recommended by Parnin and Orso [27]. A higher rank of faulty statements means better fault localization performance.

[1]http://sir.unl.edu/portal/index.php

C. Results and Analysis

In order to compare the performance between our approach `FLBF` and the six typical fault localization techniques using binary execution information, our study analyzes the experiment results from two aspects: the boxplots and Wilcoxon-signed-rank testing.

Boxplots We first leverage boxplots of experimental results to demonstrate the effectiveness of branching execution probability in input information and compare the performance of our approach `FLBF` with six typical fault localization techniques [28]. Fig. 4 displays boxplots of faulty statements' absolute ranks under each fault localization technique against different subject programs. In every boxplot, a specific technique on the horizontal axis corresponds to two boxes: the left box stands for the first scenario and the right box stands for the second scenario. We can notice that after adding branching execution probability into input information, every fault localization technique in this experiment experiences an integral improvement of result compared with the original situation. Moreover, it is obvious to notice that `FLBF` has higher rank, narrower range of variation and more stable performance as compared with the results of the other six fault localization techniques, even if in the first scenario where the input is binary execution information . Here we take the boxplots of the program `space` under the second scenario as example. With higher ranks, Jaccard, GP01, GP02 and `FLBF` perform better as compared with the other ones in `space`. Meanwhile, `FLBF` has the highest average rank and the smallest range of rank variation. More specifically, the average rank of `FLBF` arrives at slightly over 1500 whereas the average ranks of Jaccard, GP01 and GP02 are all around 2000. Therefore, `FLBF` has a better performance than the other fault localization techniques in the program `space`. Based on the boxplots in Fig. 4, `FLBF` can significantly increase the absolute rank of the faulty statement in each formula of all subject programs, and thus narrow down the searching domain of faulty statements.

Statistical comparison Although the boxplots provide a direct visual comparison between `FLBF` and the six typical fault localization techniques, a quantitative evaluation is still indispensable. Therefore, we further conduct a more scientific and rigorous method, that is, the paired Wilcoxon-Signed-Rank Test to evaluate the effectiveness of `FLBF` over that of the other fault localization techniques. The paired Wilcoxon-Signed-Rank test is a non-parametric statistical hypothesis test for testing that the differences between pairs of measurements $F(x)$ and $G(y)$, which do not follow a normal distribution [29]. It makes use of the sign and the magnitude of the rank of the differences between $F(x)$ and $G(y)$. At the given significant level σ, we can use both 2-tailed and 1-tailed p-value to obtain a conclusion.

The experiments performed one paired Wilcoxon-Signed-Rank test: the localization effectiveness of `FLBF` v.s. that of the other six fault localization techniques all in the second scenario. Each test uses both the 2-tailed and 1-tailed checking at the σ level of 0.05. Given a program, we use the list of the ranks of the faulty statement in all faulty versions of the program

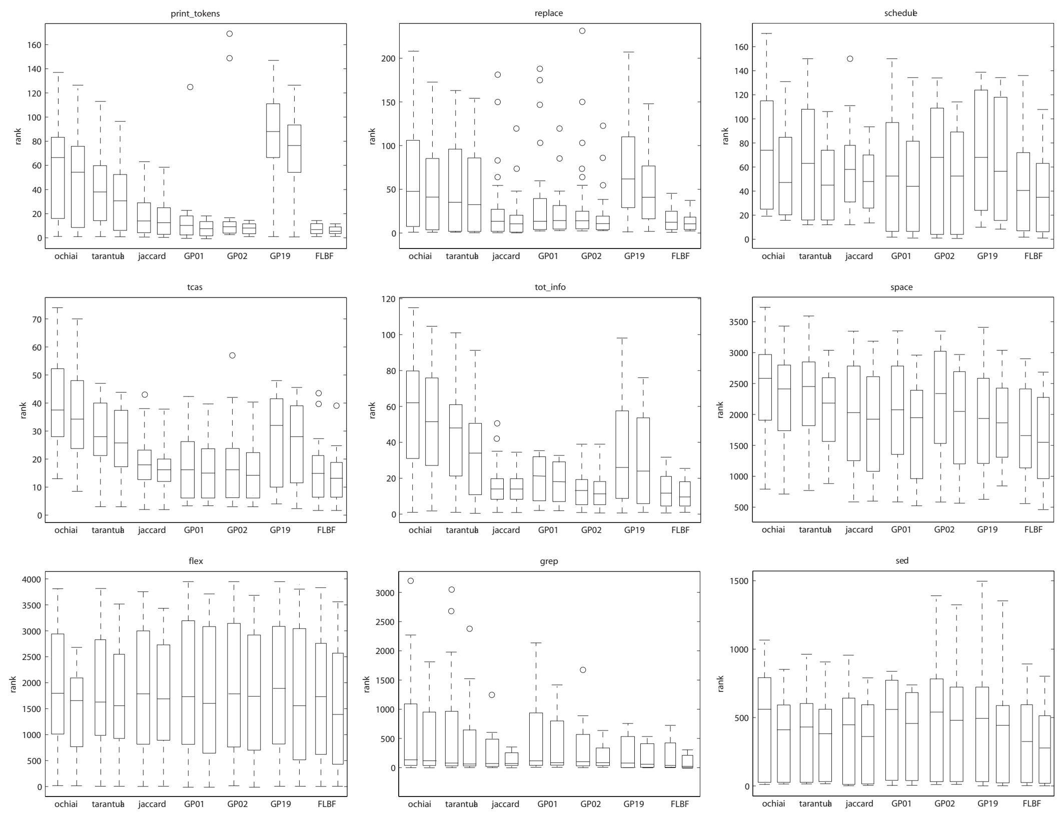

Fig. 4. Boxplots of the experiment results.

for using our approach FLBF as the list of measurements of $F(x)$, while the list of measurements of $G(y)$ is the list of the ranks of the faulty statement for using one of the other six fault localization techniques. Hence, in the 2-tailed test, FLBF has SIMILAR effectiveness as the compared fault localization technique when the null hypothesis H_0 is accepted at the significant level of 0.05. And in the 1-tailed test (right), FLBF has WORSE effectiveness than the compared fault localization technique when the alternative hypothesis H_1 is accepted at the significant level of 0.05. Finally, in the 1-tailed test (left), FLBF has BETTER effectiveness than the compared fault localization technique when H_1 is accepted at the significant level of 0.05.

TABLE III shows the statistical results of FLBF over each of six typical fault localization techniques in each subject program. The "total" row demonstrates the statistical comparison between the ranks of the faulty statements in all faulty versions of all subject programs using FLBF and those using each of the six fault localization techniques. As shown in Table III, FLBF obtains BETTER results over the six typical fault localization techniques almost on all the subject programs. For example, in the subject program *replace* FLBF performs BETTER than the six fault localization techniques through both 2-tailed and 1-tailed p-value. However we also notice that there are six SIMILAR results in TABLE III: FLBF v.s. GP02 (in *print_tokens*, *schedule* and *grep*), FLBF v.s. GP01 (in *tcas* and *tot_info*) and FLBF v.s. Jaccard (in *grep*). Since the six classic techniques and FLBF realize fault localization from different empirical analysis: the former ones focus on the statement execution and the latter puts emphasis on the data relevance, they both have deviation from the program's real data flow and control flow. On the other hand, the result indicates that FLBF, at least, has reached the same performance as GP01, GP02 and Jaccard which are the most effective fault localization techniques in recent researches [19]. In addition, FLBF also obtains BETTER results on "total" comparison over the six fault localization techniques. The results identify that the absolute rank produced by FLBF significantly tends to be less than the one using the six typical fault localization techniques, that is, FLBF performs significantly better than the six representative fault localization techniques based on these subject programs.

D. Threats to Validity

In this section, we summarize the threats to validity of our study including but not limited to the following three aspects: threats to internal validity, threats to external validity, and threats to construct validity.

Threats to internal validity This type of threats involves the relationship between independent and dependent variables in this study which are beyond researchers' knowledge. There may be chances that some undetected implementation flaws existing in our experiment may have affected the results. To ensure the accuracy of the experiments, we have carefully

TABLE III
Statistical comparison of FLBF and typical fault localization techniques.

Program	Comparison	2-tailed	1-tailed(right)	1-tailed(left)	Conclusion
print_tokens (2 ver.)	FLBF v.s. Ochiai	2.37E-03	9.99E-01	1.33E-03	BETTER
	FLBF v.s.Tarantula	2.65E-03	9.99E-01	1.48E-03	BETTER
	FLBF v.s.Jaccard	3.90E-02	9.82E-01	2.12E-02	BETTER
	FLBF v.s.GP01	1.69E-02	9.22E-01	9.07E-02	BETTER
	FLBF v.s.GP02	5.93E-01	7.19E-01	5.12E-02	SIMILAR
	FLBF v.s.GP19	1.87E-03	9.99E-01	1.05E-03	BETTER
replace	FLBF v.s. Ochiai	7.49E-04	1.00E+00	3.91E-04	BETTER
	FLBF v.s.Tarantula	9.59E-04	1.00E+00	4.99E-04	BETTER
	FLBF v.s.Jaccard	4.02E-02	8.03E-01	2.04E-01	BETTER
	FLBF v.s.GP01	4.32E-02	7.52E-01	2.55E-01	BETTER
	FLBF v.s.GP02	8.49E-03	5.81E-01	4.29E-03	BETTER
	FLBF v.s.GP19	1.22E-05	1.00E+00	6.51E-06	BETTER
schedule (2 ver.)	FLBF v.s. Ochiai	1.56E-02	9.93E-01	8.30E-03	BETTER
	FLBF v.s.Tarantula	2.22E-02	9.90E-01	1.18E-02	BETTER
	FLBF v.s.Jaccard	1.27E-02	9.39E-01	6.64E-02	BETTER
	FLBF v.s.GP01	1.33E-02	9.36E-01	6.93E-02	BETTER
	FLBF v.s.GP02	1.84E-01	9.12E-01	9.56E-02	SIMILAR
	FLBF v.s.GP19	2.06E-02	9.01E-01	1.07E-02	BETTER
tcas	FLBF v.s. Ochiai	8.06E-07	1.00E+00	4.19E-07	BETTER
	FLBF v.s.Tarantula	6.13E-05	1.00E+00	3.17E-05	BETTER
	FLBF v.s.Jaccard	9.00E-03	9.96E-01	4.60E-03	BETTER
	FLBF v.s.GP01	2.27E-01	8.88E-01	1.15E-01	SIMILAR
	FLBF v.s.GP02	9.41E-03	9.54E-01	4.78E-03	BETTER
	FLBF v.s.GP19	1.00E-05	1.00E+00	5.19E-06	BETTER
tot_info	FLBF v.s. Ochiai	2.54E-04	1.00E+00	1.38E-04	BETTER
	FLBF v.s.Tarantula	5.35E-04	1.00E+00	2.90E-04	BETTER
	FLBF v.s.Jaccard	1.27E-02	9.94E-01	6.81E-03	BETTER
	FLBF v.s.GP01	1.93E-01	9.08E-01	1.00E-01	SIMILAR
	FLBF v.s.GP02	9.06E-03	5.57E-01	4.62E-03	BETTER
	FLBF v.s.GP19	1.07E-03	1.00E+00	5.79E-04	BETTER
space	FLBF v.s. Ochiai	1.04E-03	9.99E-01	5.42E-04	BETTER
	FLBF v.s.Tarantula	3.56E-03	9.98E-01	1.85E-03	BETTER
	FLBF v.s.Jaccard	1.08E-02	9.95E-01	5.56E-03	BETTER
	FLBF v.s.GP01	1.31E-02	9.94E-01	6.74E-03	BETTER
	FLBF v.s.GP02	1.96E-03	9.99E-01	1.02E-03	BETTER
	FLBF v.s.GP19	3.83E-03	9.98E-01	1.98E-03	BETTER
flex	FLBF v.s. Ochiai	1.52E-01	9.25E-01	7.64E-02	BETTER
	FLBF v.s.Tarantula	9.29E-02	4.66E-02	5.37E-01	BETTER
	FLBF v.s.Jaccard	2.17E-02	8.92E-01	1.09E-02	BETTER
	FLBF v.s.GP01	1.53E-06	1.00E+00	7.83E-07	BETTER
	FLBF v.s.GP02	2.60E-04	1.00E+00	1.32E-04	BETTER
	FLBF v.s.GP19	4.86E-09	1.00E+00	2.49E-09	BETTER
grep	FLBF v.s. Ochiai	2.87E-01	8.62E-01	1.49E-01	BETTER
	FLBF v.s.Tarantula	4.92E-01	7.61E-01	2.54E-01	BETTER
	FLBF v.s.Jaccard	6.79E-01	6.70E-01	3.49E-01	SIMILAR
	FLBF v.s.GP01	6.20E-03	9.80E-01	3.23E-03	BETTER
	FLBF v.s.GP02	8.13E-01	6.03E-01	4.16E-01	SIMILAR
	FLBF v.s.GP19	2.66E-02	9.36E-01	1.38E-02	BETTER
sed	FLBF v.s. Ochiai	3.25E-04	1.00E+00	1.71E-04	BETTER
	FLBF v.s.Tarantula	4.62E-03	9.98E-01	2.41E-03	BETTER
	FLBF v.s.Jaccard	4.87E-02	9.76E-01	2.52E-02	BETTER
	FLBF v.s.GP01	2.23E-02	9.89E-01	1.16E-02	BETTER
	FLBF v.s.GP02	1.05E-02	9.95E-01	5.49E-03	BETTER
	FLBF v.s.GP19	9.04E-03	9.96E-01	4.70E-03	BETTER
total	FLBF v.s. Ochiai	1.31E-13	1.00E+00	6.60E-14	BETTER
	FLBF v.s.Tarantula	1.39E-08	1.00E+00	6.95E-09	BETTER
	FLBF v.s.Jaccard	6.07E-05	1.00E+00	3.04E-05	BETTER
	FLBF v.s.GP01	1.29E-10	1.00E+00	6.48E-11	BETTER
	FLBF v.s.GP02	1.92E-07	1.00E+00	9.64E-08	BETTER
	FLBF v.s.GP19	2.54E-21	1.00E+00	1.27E-21	BETTER

realized the relevant techniques and comprehensive functional testing in this article.

Threats to external validity This type of threats corresponds to the generalization of the experimental results. The threat of external validity is about the subject programs. Aiming at obtain credible experimental results, we select two standard benchmarks (`Siemens` and `space`) and three real-life UNIX utility programs (`flex`, `grep` and `sed`) as our subject programs because they are widely used in the field of fault localization. However, the type of all faults in these programs is single-fault. From the experience of real-life projects, a faulty program may have multiple faults at the same time. For multiple faults, we can apply the clustering technology(*e.g.* [6]) to transform the context of multiple faults into the same kind of single faults, and thus our approach can be applicable to multiple faults. In addition, the research [30] has shown that multiple faults usually pose a negligible effect on the effectiveness of fault localization in spite of the effect of fault localization interference. These findings increase our confidence of the experimental results in the context of multiple faults. Even so, in the realistic debugging, researchers may encounter many unknown and complicated situations. Therefore, it is essential to use more real-life subjects programs (such as multiple-faults programs and large-sized programs) to further justify the experimental results.

Threats to construct validity This type of threats concerns the appropriateness of the evaluation measurement. We use the rank of the faulty statement in the ranking list to evaluate the effectiveness of fault localization techniques. This metric is highly recommended by the recent research [31] [32] and thus the threat is acceptably mitigated.

V. Conclusion

The huge demand of debugging work from real life is driving the study of fault localization and development of different techniques. Many current fault localization techniques basically depend on the binary execution information which is the information of each program statement being executed or not executed by a particular test case. However, this simple information may miss some essential clues such as the branching executing information. To alleviate this problem, this paper proposes a fault localization approach called `FLBF` which utilizes the branching execution information in the programs and adopts one of the most widely used feature selection method called Fisher score to rank the suspicious statements. The scenario used to demonstrate the utility of FLBF is composed of two standard benchmarks and three real-life UNIX utility programs. and then we compare FLBF with other six typical fault localization techniques based on them. In order to present our experiments in a scientific and rigorous way, we conduct both the boxplots analysis and Wilcoxon-Signed-Rank Testing to justify the advantages of our method. The experimental results show that input with branching execution information can improve the performance of current fault localization techniques and FLBF performs more stably and efficiently than other six typical fault localization techniques.

As for future work, first we plan to enrich our study by taking more subtle information in the programs into account such as the looping execution information. After this, we plan to implement our method on more complex real-life software projects. This is necessary because many current fault localization techniques fail to provide stable and effective solution for those complex programs and the industry has a huge requirement of truly useful fault localization techniques.

Acknowledgment

This research was supported by the National Natural Science Foundation of China under Grant (Nos. 61379054 and 91318301).

References

[1] A. Zeller, *Why Programs Fail: A Systematic Guide to Debugging*. Morgan Kaufmann, 2005.

[2] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. ACM, 2005, pp. 342–351.

[3] W. Jin and A. Orso, "F3: fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, 2013, pp. 213–223.

[4] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. ACM, 2002, pp. 467–477.

[5] R. Abreu, P. Zoeteweij, and A. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. IEEE, 2007, pp. 89–98.

[6] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA 2007)*. ACM, 2007, pp. 16–26.

[7] L. Naish, H. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, p. 11, 2011.

[8] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, 2014.

[9] X. Xie, T. Y. Chen, F.-C. 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)*, 2013.

[10] C. Sun and S.-C. Khoo, "Mining succinct predicated bug signatures," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE2013)*. ACM, 2013, pp. 576–586.

[11] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OPPSLA 2013)*. ACM, 2013, pp. 765–784.

[12] Y. Lei, X. Mao, Z. Dai, and D. Wei, "Effective fault localization approach using feedback," *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 9, pp. 2247–2257, 2012.

[13] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *the 28th International Conference on Automated Software Engineering (ASE 2013)*, 2013, pp. 257–267.

[14] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," *Acm Sigplan Notices*, vol. 33, no. 7, pp. 83–90, 1997.

[15] G. Isabelle and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[16] L. Y. Qi Y, Mao X, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 191–201.

[17] T. B. Le and D. Lo, "Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM 2013)*, 2013, pp. 310–319.

[18] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*. IEEE, 2002, pp. 595–604.

[19] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Proceedings of 4th International Symposium on Search-Based Software Engineering (SSBSE 2012)*, 2012, pp. 244–258.

[20] P. Duda and D. G. Stork, *Pattern Classification*. Wiley-Interscience Publication, 2001.

[21] E. A. Guyon I, "An introduction to variable and feature selection," *The Journal of Machine Learning Research*, pp. 1157–1182, 2003, 3.

[22] M. H. Liu H, "Feature selection for knowledge discovery and data mining," *Springer Science & Business Media*, 2012.

[23] K. J. Pudil P, Novovičová J, "Floating search methods in feature selection," *Pattern recognition letters*, vol. 15(11):, pp. 1119–1125, 1994.

[24] Q. Gu, Z. Li, J. Han, Q. Gu, and Z. Li, "Generalized fisher score for feature selection," *Uai*, 2012.

[25] W. M. Ahn S, Korattikara A, "Bayesian posterior sampling via stochastic gradient fisher scoring," *arXiv preprint arXiv*, 2012.

[26] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.

[27] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. ACM, 2011, pp. 199–209.

[28] I. B. Frigge M, Hoaglin D C, "Some implementations of the boxplot," *The American Statistician*, vol. 43(1), pp. 50–54, 1989.

[29] G. W. Corder and D. I. Foreman, *Nonparametric statistics for non-statisticians: A step-by-step approach*. John Wiley & Sons, 2009.

[30] N. DiGiuseppe and J. Jones, "On the influence of multiple faults on coverage-based fault localization," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. ACM, 2011, pp. 210–220.

[31] J. LI, Q. TAN, and L. TAN, "Implementing low-cost fault tolerance via hybrid synchronous/asynchronous checks," *Journal of Circuits System & Computers*, vol. 22, no. 7, pp. 1332–1346, 2013.

[32] J. Li, J. Xue, X. Xie, Q. Wan, Q. Tan, and L. Tan, "E pipe : A low-cost fault-tolerance technique considering wcet constraints," *Journal of Systems Architecture*, vol. 59, no. 10, pp. 1383–1393, 2013.