Software Fault Localization: Techniques, Issues and Remedies

Amol Saxena, Roheet Bhatnagar, and Devesh Kumar Srivastava

Abstract— **Software fault localization is a task of isolating the statements which cause faults in a program. Fault localization is one of the monotonous, time consuming and prohibitively expensive, yet very important activities in program debugging. Manual testing and debugging is very infeasible due to the escalating scale and complexity of software systems. So, there is a strong need for automated techniques which can help developers locating bugs in programs without much human interference. This necessity has given rise to the development of a variety of fault localization techniques, each of which deals with the problem in its own way. This paper presents an overview of such techniques with some key issues and concerns relevant to software fault localization. In particular, this paper focuses on spectrum-based software fault localization (SBFL) techniques and reviews two recent approaches in detail that further improve its performance. These two approaches which are based on the concepts of Failed Execution Slice and Fault Context are evaluated experimentally on seven standard benchmark Siemens programs to compare their effectiveness against the classic Ochiai method. The experimental results show that the two approaches improve SBFL performance by an average of 27.05% and 38.64% respectively against the classic Ochiai technique.**

Index Terms— **Software fault localization, execution trace, debugging, failure, program spectrum, program slicing, failed execution slice, fault context.**

I. Introduction

Today the influence of software is reasonably everywhere. At the present time, software is key element to many systems and processes from the safety point of view such as healthcare, aeronautics, industrial plants, nuclear energy etc. This development has been continuously increasing the scale and complexity of software systems day by day. Unfortunately it has resulted in many software bugs that remain undetected during the development process and ultimately passes to the end user which may result in huge losses because of failures. The significant proportion of cost of fixing software bugs is passed to the software users and rest is absorbed by the developers and vendors.

The objective of fault localization is to identify the defective program elements which lead to software failures. In other words, fault localization is the process to identify the locations of faults in software programs. Previously the task of fault localization was performed manually which was very tedious, time consuming and prone to failures as many bugs remain undetected. This manual process of localizing faults in today's large scale complex and safety critical software systems is prohibitively expensive. Another problem of manual fault detection is that it depends on experience, judgment and perception of developers and testing engineers to identify the code that causes software failure. These limitations have given rise to the requirement of developing more scientific techniques for fault localization. It is also important to develop techniques that can fully or partially automate the task of identification of faulty code in software systems. The research is continuously going on in this direction and many techniques and concepts have been developed that are helping software professionals to improve the quality of the software and to improve the bug localization process. As advances are being made from both theoretical and practical perspective in the field of software fault localization, it is important to give an overview of current techniques related to fault localization to facilitate those who want to contribute in this area. In software fault localization literature many studies have been proposed that further improve the performance of existing classical fault localization methods. This paper describes two such techniques in detail and experimentally evaluates their effectiveness against the classic SBFL technique (Ochiai in our study).

It is necessary to give brief definitions of the following terms which appear frequently in this paper. A failure occurs when a service differ from its accepted behavior. An error can be defined as a state or condition that may cause a failure and a fault or bug is the primary source of an error [1]. The main contributions of this paper are summarized below.

- A review of the basic and advanced software fault localization techniques and examining their issues and concerns.
- This paper illustrates the traditional spectrum-based fault localization technique and some of the recent methods that further improve its performance.
- Provides an overview of some standard metrics that are used to evaluate the effectiveness of software fault localization techniques.

Manuscript received June 28, 2021; revised February 9, 2022.

Amol Saxena is a PhD candidate of Department of Computer Sc. Engineering, SCIT, Manipal University Jaipur, Jaipur, Rajasthan, India-303007 (Phone: +91-9982776883, email: amolsaxena2015@gmail.com)

Dr. Roheet Bhatnagar is a Professor of Computer Sc. & Engineering Department, SCIT, Manipal University Jaipur, Jaipur, Rajasthan, India-303007 (email: roheet.bhatnagar@jaipur.manipal.edu)

Dr. Devesh Kumar Srivastava is a Professor of Information Technology Department, SCIT, Manipal University Jaipur, Jaipur, Rajasthan, India-303007 (email: devesh988@yahoo.com)

- Experimentally evaluates two recent techniques on a standard benchmark (Siemens test suite) to compare their effectiveness against the classic SBFL method Ochiai.

The rest of the paper is organized as follows: the second section briefly discusses about the conventional fault localization techniques, and then categorization of more advanced techniques of fault localization is presented in third section. This section illustrates the traditional methods of spectrum-based fault localization and also illustrates some recent techniques by which the effectiveness of spectrum-based fault localization can be improved. The first technique discussed here is based on the failed execution slice and the second method utilizes the concept of fault context in SBFL to improve the absolute rank of faulty program entities. A brief review of other software fault localization technique is also provided in section three. An overview of various evaluation metrics that are used to measure the effectiveness of fault localization techniques is presented in section four. The section five presents a detailed experimental study to evaluate the effectiveness of the two techniques (i. e. failed execution slice and fault context) against the classic SBFL method Ochiai. In the end conclusion is presented in section six.

II. Conventional Fault Localization Techniques

To give a basic overview of fault identification, this section explains some traditional and instinctive techniques of fault localization.

A. Program Logging

To monitor variable values and other state information of the program, statements such as print are inserted into the code. This creates a program log, which is used by developers to detect the underlying cause of failure in case of abnormal program behavior is detected.

B. Assertions

Assertions are added by developers in the form of constraints which are required to be always true during the correct execution of a program code. Assertions are specified in the program as conditional statements that terminate the program if evaluated to false. In this way, incorrect or faulty execution of a program can be detected by assertions.

C. Breakpoints

With the help of breakpoints the user can temporarily stop the execution of a program when it reaches a certain point and thus allows a user to examine the current state of variables and intermediate results. User can observe development of a bug after a breakpoint is activated. This approach is adopted by tools for example GNU GDB and Microsoft Visual Studio Debugger.

D. Profiling

In order to optimize a program, profiling can be used by analyzing run time metrics such as memory usage and execution speed. Profiling is helpful in debugging in the following manner – detecting when a function execute unexpectedly and identifies the code responsible for that, discovering the state of memory leak and investigating the side effects of lazy evaluation i.e. evaluation of expressions is deferred until some other computation is awaiting their results. The examples of some debugging tools that incorporate profiling are GNU's "gproof" and the Eclipse plug-in "TPTP".

III. Advanced Fault Localization Techniques

As the size and complexity of software system is increasing continuously, traditional fault identification techniques are insufficient to detect the root cause of failures. This section discusses different category of fault localization techniques. The authors illustrate the differences between different slice-based techniques first, and then this paper presents spectrum-based techniques with the help of an example. The spectrum-based techniques are commonly used in fault localization to compute suspiciousness values of program statements to identify the location of faults that are responsible for program failure. Many improvements in spectrum-based methods have been proposed by various authors time to time and two such methods that improve the performance of spectrum-based fault localization are explained in this paper with the help of examples. Next, we give a brief overview of some other techniques such as statistics-based, program state-based, machine learning based, data mining-based and model-based techniques.

A. Slice-Based Techniques

Program slicing is a method or approach that conceptualizes a program into a compact manner by removing irrelevant parts which have no effect upon the semantics of interest. Program slicing only focuses on selected aspects of semantics. Program slicing reduces the search domain while developers locate faults in a program. The idea is that when failures occurs in a program because of an erroneous variable value at a statement, then the defect is there in the static slice related to the variable statement pair which restricts the search efforts to the particular slice rather than the entire program. One limitation with static slicing technique is that it does not work well with pointer variables because pointers make data flow analysis inefficient as dereferencing of pointer variables introduces large sets of data facts which need to be stored. Equivalence analysis improves effectiveness of data flow analysis while working with pointer variables. It finds equivalence relationship between memory locations accessed by a program segment. When two memory locations are equal then they share same data objects in a function or procedure. Thus, it is required to figure out information for a representative memory location by data flow analysis, and data flow for other locations can be acquired from the representative location.

Static slicing has a drawback that the slice for a given variable at a given program statement contains all the executable statements of the program that could somehow affect the value of this variable of the slice. Consequently, it might include some extra unnecessary statements because run time values cannot be predicted with static slicing method. To keep out such extra statements dynamic slicing should be used. The dynamic slicing is constructed with respect to the conventional static slicing criterion together with the input sequence supplied to the program, during

TABLE I
COMPARISON OF DIFFERENT TYPES OF SLICING METHODS

Stmt. #	Code snippet with a bug at statement S15	Static slice for maximum	Dynamic slice for maximum w. r. t. test case a=1, b=2, c=3	Execution slice for maximum w. r. t. test case a=1, b=2, c=3
S1	input (a);	Enter (a);	Enter (a);	Enter (a);
S2	input (b);	Enter (b);	Enter (b);	Enter (b);
S3	input (c);	Enter (c);	Enter (c);	Enter (c);
S4	int maximum;	int maximum;	int maximum;	int maximum;
S5	int minimum;			int minimum;
S6	if(a>b)	if(a>b)	if(a>b)	if(a>b)
S7	if(a>c){	if(a>c){		
S8	maximum=a;	maximum=a;		
S9	if(b>c)			
S10	minimum=c;			
S11	else			
S12	minimum=b;}			
S13	else	else		
S14	{ maximum=c;	{ maximum=c;		
S15	minimum=a; //correct minimum=b;}			
S16	else if(b>c) {	else if(b>c) {	else if(b>c) {	else if(b>c) {
S17	maximum=b;	maximum=b;		
S18	if(a>c)			
S19	minimum=c;			
S20	else			
S21	minimum=a;}			
S22	else{ maximum=c;	else{ maximum=c;	else{ maximum=c;	else{ maximum=c;
S23	minimum=a;}			minimum=a;}
S24	print(maximum);	print(maximum);	print(maximum);	print(maximum);
S25	print(minimum);			print(minimum);

some specific execution (dynamic information). One weak point with dynamic slicing is that it may omit some crucial statements which can lead to failure. The dynamic slicing cannot find the execution omission errors which may cause to leave out some significant statements in a program and therefore resulting in failure. The relevant slicing concept can be used to trace such execution omission errors by locating faulty statements.

A third type of slicing is execution slicing which includes statements that execute with respect to a given test case. It gives statements executed related to a specific inputs as against the case of static slicing which considers all inputs. Some of the execution slice based debugging tools in practice are xSuds at Telcordia, and eXVantage at Avaya. The following example given in Table I demonstrates the difference between static, dynamic and execution slicing.

Assume the sample code given in the second column of Table I has a bug at statement S15. The third column shows the static slice of variable *maximum.* This static slice contains all statements of the program that might influence the value of the *maximum* variable. The statements included in this static slice are S1, S2, S3, S4, S6, S7, S8, S13, S14, S16, S17, S22 and S24. The fourth column shows the dynamic slice for variable *maximum* which includes statements that affect the value of *maximum* with respect to test case when a=1, b=2 and c=3. This dynamic slice is consists of statements S1, S2, S3, S4, S6, S16, S22, and S24.The execution slice with respect to test case a=1, b=2 and c=3 is given in column five and contains all statements of the program executed by this test case. This execution slice contains statements S1, S2, S3, S4, S5, S6, S16, S22, S23, S24, and S25. Slices are difficult to understand due to their length. The notation of barriers is proposed by Krinke [2], which offers a filtering method for smaller program slices and improved ability to understand.

B. Program Spectrum-Based Techniques

A program spectrum explains the execution details from certain viewpoints, such as conditional branching or loop free paths within procedures. The use of the program spectrum techniques in software fault localization was suggested by Collofello and Cousins [3]. Such spectra information can be used to identify suspicious code which is responsible for program crash. The program segment under testing during execution is denoted by Executable Statement Hit Spectrum (ESHS). This information enables to identify components of a program involved in a failure.

i. Notations

Some notations that are used in spectrum-based techniques are defined here. P represents a program. N_{CF} means "number of failed test cases that cover a program statement"; N_{CS} signifies the "count of successful test cases that cover a statement", N_C symbolizes the "total number of test cases by which a statement is covered", N_S and N_F represent the "total number of successful and failed test cases" respectively. N_{UF} means the "count of failed test cases that do not cover the statement", and N_{US} specifies the "number of successful test cases that do not cover the statement".

ii. Techniques

For spectrum-based fault localization, some early researches only used failed test cases. These studies found to be ineffective. Afterwards researches used both successful and unsuccessful test cases and highlighted the differences between them and achieved better fault localization results. M. Renieris et al. [4] worked on methods known as set union and set intersection. In set union algorithm program

spectra differences are determined between a failing test case f and a set of successful test cases P. Consider S (t) as the spectra or in other words running behavior of the program executing the test case t. Then, A fault localization report R is generated by determining the differences between a failing test case f, and all passed test cases $p_i \in P$. This can be represented as R= S (f) – $U_{pi} \in p$ S(pi). This algorithm is known as set union algorithm and is described in [4]. The code executed by failed test cases but not successful test cases is more suspicious than others. This is the technique of set union methods which focuses on source code that is executed by failed test cases but not by any of the test cases that successfully executed the code.

As opposed to above Set Union algorithm, a different algorithm known as Set Intersect is also explained in. As per set intersect method, the program spectra difference between failing test case and intersection spectra of successful test cases is computed as localization report, R. This can be written as $R = \cap p_i \in p\ S(p_i) - S(f)$. The code that is executed by all the successful test cases but not failed tests is excluded by set intersection method.

Another program spectrum-based technique is nearest neighbor, in which a successful test run is find out that is most similar to the failed tests based on distance metric. The difference set of this successful and failed test is computed that locates the fault is it is present in the difference set.
The thought of nearest neighbor resembles counterfactual reasoning, which says that, suppose there are two events X and Y, in world 'w' and X causes Y if, in all possible worlds that are similar to 'w', X does not occur and Y also does not happen.

Next, the Tarantula, a well-known ESHS based similarity coefficient based technique is discussed. This technique makes use of coverage and execution results of a program under test. The execution results tell the success and failure of the program and suspiciousness of each statement (S) is computed as per the formula given below in (1). The notations used here are explained in the previous section.

$$\text{susp (s)} = \frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}} \quad (1)$$

A study shows that, in comparison to set union, set intersection and nearest neighbor methods, the technique known as Tarantula is relatively more efficient fault localization technique because before locating the first faulty statement in a program, Tarantula checks fewer code statements [5]. Those statements which are executed by same number of failed test cases are grouped together and these groups are given ranks by the number of failed test cases, and are arranged in descending order. Statements are ranked on the basis of suspiciousness calculated within each group.

Now, the authors give an example to compute the suspiciousness values of statements of a program segment using the Tarantula Technique [1]. Consider the code snippet given in Table II. Here it is assumed that there are five successful test cases (t1, t2, t4, t5 and t6) and one unsuccessful or failed test case (t3).

The statement coverage of six test cases is shown from third to eighth columns. The bottom row gives the execution result of each test case. Here, '0' means successful execution of the test case and '1' represents failed test case execution. The entry in the table with an '1' shows that the corresponding test case covers the statement, while an empty entry means that the statement is not covered. The next two columns contain the values of N_{CF} and N_{CS} for each statement of the program code. The suspiciousness value is displayed in the eleventh column as per the definition of Tarantula. The last column displays the ranking of each statement. It can be observed that the faulty statement S13 has the maximum ranking that is 1. It requires only two searches to detect the faulty statement S13 which has the highest suspiciousness rank of 1. As per EXAM score only 9.09% of statements need to be searched in order to reach to the faulty statement. Some other recent techniques have given better performance in terms of their usefulness at fault localization. The technique based on

TABLE II
SUSPICIOUSNESS VALUE OF PROGRAM STATEMENTS COMPUTED USING THE TARANTULA METHOD

Stmt. #	Program	t1	t2	t3	t4	t5	t6	N_{CF}	N_{CS}	Suspiciousness	Ranking
S1	input (a);	1	1	1	1	1	1	1	5	0.50	4
S2	input (b);	1	1	1	1	1	1	1	5	0.50	4
S3	input (c);	1	1	1	1	1	1	1	5	0.50	4
S4	int maximum;	1	1	1	1	1	1	1	5	0.50	4
S5	int minimum;	1	1	1	1	1	1	1	5	0.50	4
S6	if(a>b)	1	1	1	1	1	1	1	5	0.50	4
S7	if(a>c){	1	1	1				1	2	0.71	3
S8	maximum=a;	1	1					0	2	0.00	12
S9	if(b>c)	1	1					0	2	0.00	12
S10	minimum=c;	1						0	1	0.00	12
S11	else minimum=b;}		1					0	1	0.00	12
S12	else { maximum=c;			1				1	0	1.00	1
S13	minimum=a; //correct minimum=b;}			1				1	0	1.00	1
S14	else if(b>c) {				1	1	1	0	3	0.00	12
S15	maximum=b;				1	1		0	2	0.00	12
S16	if(a>c)				1	1		0	2	0.00	12
S17	minimum=c;				1			0	1	0.00	12
S18	else minimum=a;}					1		0	1	0.00	12
S19	else{ maximum=c;						1	0	1	0.00	12
S20	minimum=a;}						1	0	1	0.00	12
S21	print(maximum);	1	1	1	1	1	1	1	5	0.50	4
S22	print(minimum);	1	1	1	1	1	1	1	5	0.50	4
Execution results (0=successful, 1=Failed)		0	0	1	0	0	0				

similarity coefficient given by Ochiai [1] is more effective than Tarantula. Its formula is given below in (2).

$$susp\ (Ochiai) = \frac{N_{CF}}{\sqrt{N_F \text{ x } (N_{CF} + N_{CS})}} \quad (2)$$

Naish et al. [6] worked on fault localization techniques for programs having single and multiple bugs. Two such techniques are O and O^P respectively. Experimental data suggest both O and O^P both are more efficient methods in comparison with Tarantula and Ochiai as far as single bug programs are concerned.

It was also found that O is a better approach for single bug programs and O^P is good for programs which have multiple bugs. The technique O (ranking metric) characterizes the suspiciousness of a statement S as given below in (3).

$$susp(s) = \begin{matrix} -1 \\ N_{US,} \end{matrix} \quad \begin{matrix} if\ N_{UF} > 0 \\ \text{otherwise} \end{matrix} \quad (3)$$

Here it is assumed that, with respect to single bug programs, N_{UF} is always zero for the faulty statements. For statements with positive suspiciousness values, the probability of them being faulty is relative to N_{US}. The technique O^P was proposed for better performance in case of programs with multiple bugs. It suggests that the statements with larger N_{CF} and smaller N_{CS} are moved to the top ranking with the help of the following (4).

$$\text{susp}(s) = N_{CF} - \frac{N_{CS}}{N_S + 1} \quad (4)$$

The use of data flow spectra in improving effectiveness of spectrum-based fault localization is given by Ribeiro et al. [7]. The data flow in a program means defining a variable i.e. assigning value to it and its subsequent references i.e. the use of that variable takes place when its value is referred to. The data flow spectrum is related to all such paths between every point where a variable is assigned a value and its subsequent use (p-use, predicate computation and c-use, value computation). This is also called definition-use associations (DUA). Ribeiro et al. [7] carried out experiments and compare the use of data flow spectrum (DUA) and control flow spectrum (lines) in fault localization. They concluded that data flow spectrum located more bugs and allows the programmer to examine less code than control flow (line spectrum) for different ranking metrics such as Tarantula, Ochiai, and Jaccard etc.

iii. Issues and Concerns

This section discusses some issues related to spectrum-based fault localization techniques. The contribution of failed and successful test cases is not properly explained by the spectrum-based techniques. The program statements are divided into two groups i.e. suspicious and unsuspicious groups. The statements that are executed by at least one failed test case are part of suspicious group, whereas the remaining program statements are contained in unsuspicious group. The suspicious statements are considered to be risky and for these risk factor is calculated. The unsuspicious statements are just assigned lowest values. However, the problem is that the test cases which executed successfully may contain bugs. Wong et al. [1] said that each additional test case whether successful or failed is helps in finding program bugs. As per their study, for a statement or piece of code, executed by the first failed test case, the computed suspiciousness value will be greater than or equal to the second failed test executed on that piece of code. Similarly, the suspiciousness value computed by second failed test case will be greater than or equal to the third failed test case that executes the code, and so on. If we compare the total contribution in locating program bugs by all the successful tests and all the failed tests on a piece of cede, it was found that the contribution by failed test is more than the contribution by successful test cases. Fault localization methods often compare failed tests with successful tests so it would be advantageous to know which successful test case should be chosen for comparison to reduce the search area of the fault. Wong and Qi [8] recommend that the successful test cases whose execution sequence is most similar to that of a failed one should be chosen. This similarity comparison is based on control flow based difference metric.

The ranking of the statements will be the same if they have same suspiciousness values. For example "if" statements execute in the same way and it is likely that these statements will be assigned same suspiciousness value in spectrum- based techniques. In case of these ties, besides statement coverage, additional information of frequency of statement execution is also utilized. Xu et al. [9] analyzed different tie breaking methods like confidence based methods, data dependency based and statement order based methods.

Zhao et al. [10] used the program control flow graph to study program execution because only coverage information cannot be used to analyze execution paths. They discover the relationship between failed execution and control flows and explained the mapping of distribution of failed execution to different control flows. They determined that how each block is related with failure and also verified that how a block is bug free by comparing the distributions of blocks on the same failed execution paths.

C. Techniques to Improve the Efficiency of Spectrum-Based Fault Localization

The next two sub-sections illustrate two methods that can improve the performance of the spectrum-based fault localization. The first method makes use of failed execution slice to improve the effectiveness of spectrum-based fault localization. The second method combines the suspiciousness of program entity and suspiciousness of program entity's fault context in order to improve the absolute rank of faulty entities in the program.

i. Efficient Spectrum-Based Fault Localization using Failed Execution Slice

Spectrum-based techniques of fault localization can be improved with the use of failed execution slice [11]. A set of statements executed by a test case are referred to as execution slice. An execution slice with respect to a failed test case is called failed execution slice. This method, first computes the suspiciousness score of each statement of a program under test using some existing fault localization technique such as Tarantula, Ochiai, Jaccard etc., then scope of fault is constrained using selective failed execution slice (FES) which is chosen on the basis of utility evaluation

function f.

Spectrum-based fault localization (SFL) mainly makes use of statement suspiciousness scores in order to detect faulty statements in a program. There may be certain statements that might be regarded as noise statements which have higher suspiciousness values but still they are not the main cause of the program failure. This method exploits this concept and eliminates such possible noise statements from the error report and further debugging process, thereby improving the effectiveness of SFL methods. A key failed execution slice (FES) is selected from all candidate failed execution slices to help reduce the scope of fault detection of the classical SFL methods.

a. Utility evaluation function

The primary objective is to use failed execution slice is to remove the maximum possible noise statements that have higher suspiciousness than the root cause. But it is challenging problem to identify the FES from all failed execution slices. As proposed by Shu et al. [11], the utility of failed execution slices is measured by the following function f as in (5).

$$f(Ti) = \sum_{i=1}^{n} S[Ii][i] x\ sus[i] \quad (5)$$

Where Ti is a failed execution slice, I_i is the corresponding failed test case. $S[Ii][i] \in [0,1]$ indicates that whether the statement i is executed or not by the test case I_i. Here, a 0 means the statement is not covered and 1 otherwise. n>=1 represents the number of executable statements in the program under test. The suspiciousness score of the statement i is denoted by sus[i]. The utility function value is calculated for each candidate FES and an FES with smaller utility function value is considered to be of greater use for reducing the scope of fault detection.

b. FES-based framework

The main process of FES based fault localization is shown in Fig. 1 and it consists of three broad steps as follows.

i. In the first step, the suspiciousness of statements is calculated based on a specific classic SBFL method. In our study we have used classic Ochiai method of spectrum-based fault localization. A debug report is generated which consists of all statements sorted according to their suspiciousness scores in descending order.
ii. In the second step, a key execution slice is selected which reduced the search space of faulty statements in the debug report. The key execution slice is selected by a utility evaluation function as defined as in (5).
iii. Finally, a new debugging report is generated by intersecting the failed execution slice (step-2) and the basic debug report generated in step-1. Based on this new debug report, a developer examines every statement by its suspiciousness score in descending order until the first faulty statement is found.

c. A simple illustration of fault localization using failed execution slice

To illustrate the efficiency of FES based method, consider the program given in Table III. In this example, the suspiciousness scores of statements are calculated using Tarantula technique of fault localization. The faulty statement is S6 with the suspiciousness score of 0.67. As per the example, the developer needs to examine five statements before reaching to faulty statement.

By observing Table III it is clear that there are three failed test cases t3, t5 and t6 and their related failed execution slices are T3, T5 and T6 respectively. The utility evaluation function when applied to these three FESs gives f (T3) = 5.90, f (T5) = 6.79 and f (T6) = 6.79. The T3 will be selected as key FES because it has smallest utility function value of 5.90. The original debugging report can be narrowed down by using selected key failed execution slice and a new debugging report can be obtained with the statements s1, s2, s3, s4, s5, s6, s7, s14, s15 and s20. Now, it is required examining only two statements before finding the faulty statement S6. When comparing the efficiency with the traditional spectrum-based fault localization method, it is easy to observe that there is an improvement of 60% with this new FES based method because now only two statements need to be examined as compared to five using traditional method. When considering FES T5 or T6, the reduced debug report contains statements s1, s2, s3, s4, s5, s6, s7, s8, s9, s10 and s20. In this case it requires examining 4 statements before locating the faulty statement with an improvement of 20% as compared to traditional method.

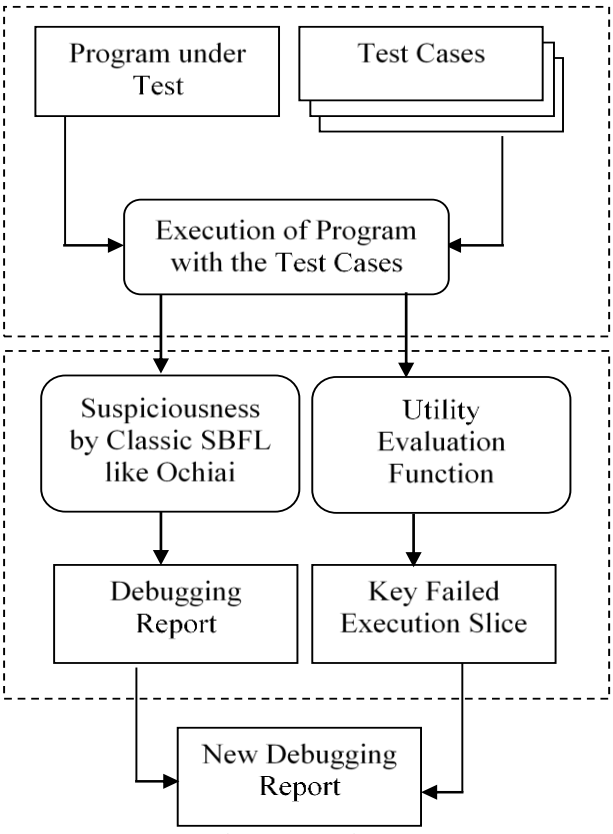

Fig. 1. Framework for FES based fault localization method

TABLE III
AN EXAMPLE SHOWING EXECUTION RESULTS OF A SAMPLE PROGRAM WITH AN ILLUSTRATION OF FAILED EXECUTION SLICES

Stmt. #	Program	t1 (1,1,1)	t2 (0,1,1)	t3 (-1, 1, 1)	t4 (-1, 0, 1)	t5 (-1,-1,-1)	t6 (-2, -1, -2)	t7 (-1, -1, 0)	N_{CF}	N_{CS}	Suspiciousness
S1	input(a)	1	1	1	1	1	1	1	3	4	0.50
S2	input(b)	1	1	1	1	1	1	1	3	4	0.50
S3	input(c)	1	1	1	1	1	1	1	3	4	0.50
S4	s=1;	1	1	1	1	1	1	1	3	4	0.50
S5	if (a<0){	1	1	1	1	1	1	1	3	4	0.50
S6	s=s * a; //correct s=s * -a;			1	1	1	1	1	3	2	0.67
S7	if (b<0){			1	1	1	1	1	3	2	0.67
S8	s = s* -b;					1	1	1	2	1	0.73
S9	if (c < 0)					1	1	1	2	1	0.73
S10	s = s * -c;					1	1		2	0	1.00
S11	else if (c >0)							1	0	1	0.00
S12	s = s+ c;										0.00
S13	else s=c;							1	0	1	0.00
S14	} else if (b > 0)			1	1				1	1	0.57
S15	s = s + b;			1					1	0	1.00
S16	else s = b;				1				0	1	0.00
S17	} else if (a> 0)	1	1						0	2	0.00
S18	s = s + a;	1							0	1	0.00
S19	else s = a;		1						0	1	0.00
S20	print (s)	1	1	1	1	1	1	1	3	4	0.50
	Execution Result (0=Successful, 1= Failed)	0	0	1	0	1	1	0			
	f(ti)			5.90		6.79	6.79				

So, the effectiveness of traditional spectrum-based fault localization methods can be improved by incorporating the concept of fault execution slices. Based on the above example, there are two important points can be concluded (1) the classic SFL methods can be improved by taking the advantage of code coverage information extracted from failed execution slices and (2) the selection of key FES depends on the utility evaluation method and can maximize the efficiency of FES based technique.

Exploring the broader applicability of this approach on different real life programs is the motivation for future work. How this approach works with other SFL techniques would also be interesting to note.

ii. Spectrum-based fault localization combined with fault context

a. The concept of fault context

Spectrum-based fault localization is a practical and efficient fault localization technique. Because of its low computational overhead and ability to produce good results on large code bases it is also considered as lightweight fault localization.

Recent researches show that the spectrum-based techniques have been instrumental to locate bugs. However, the study carried out by Parnin and Orso [12] highlighted that many developers do not find these techniques very much useful if the root cause of failure is not listed in top ranked suspicious entities in the debugging reports. Therefore researchers have been working on to improve the performance of SBFL methods so that the root faults appear at higher positions in the ranking list (debugging reports) of suspicious program elements.

One such method presented by Wang et al. [13] combines the suspiciousness of program element and suspiciousness of program element's fault context in order to improve the absolute rank of faulty elements in the program. Here, the fault context of a program element means the other program elements that were executed by the same failed test case.

The concept of fault context's suspiciousness can be used in spectrum-based fault localization techniques (e.g. Ochiai, Jaccard, DStar, Tarantula etc) to improve the rank of root faults in a faulty program.

It is important to note that a program entity's overall suspiciousness rank will be higher if the suspiciousness of that program entity is higher and the suspiciousness of its fault context is lower. To illustrate this idea we use an example as given below.

b. An Illustrative Example

Consider the example program given in Table IV which counts the number of vowels, consonants, spaces and digits in a string.

There is a fault in basic block 3 (or statement 3) wherein the equality comparison in incorrectly written as line.charAt(i) == 'b'. The correct form of this comparison would be line.charAt(i) == 'a'.

There exist a total of five test cases in this example out of which two test cases execute successfully and three test cases fail that means does not produce the desired output. The program hit spectra is shown from third column to seventh column. The entry in the table with a '1' shows that the corresponding test case covers the statement, while an entry with a '0' means that the statement is not covered. The bottom row gives the execution result of each test case. Here, '0' means successful execution of the test case and '1' represents failed test case execution. The next two columns contain the values of N_{CF} and N_{CS} for each statement of the program code. These notations have been explained in section III-B. The suspiciousness value is displayed in the tenth column as per the definition of Ochiai similarity metric. The last column displays the ranking of all basic blocks in descending order based on their suspiciousness

TABLE IV
AN EXAMPLE SHOWING THE SUSPICIOUSNESS VALUE COMPUTED USING OCHIAI TECHNIQUE

Basic Block No	static void vowel(String line)	T1	T2	T3	T4	T5	N_{CF}	N_{CS}	Suspiciousness (Ochiai)	Rank
b1	{int vowels, consonant, digit, space; vowels = consonant = digit = space = 0;	1	1	1	1	1	3	2	0.77	5
b2	for (int i = 0; i <line.length() ; ++i) {	1	1	1	1	1	3	2	0.77	5
b3	if (line.charAt(i) == 'b' || line.charAt(i) == 'e' || line.charAt(i) == 'i' || line.charAt(i) == 'o' || line.charAt(i) == 'u') // correct line.charAt(i) == 'a'	1	1	1	1	1	3	2	0.77	5
b4	++vowels;	1	0	0	1	1	2	1	0.67	9
b5	else if ((line.charAt(i) >= 'a' && line.charAt(i) <= 'z'))	1	1	1	1	1	3	2	0.77	5
b6	++consonant;	1	1	1	0	1	2	2	0.58	10
b7	else if (line.charAt(i) >= '0' && line.charAt(i) <= '9')	0	0	1	1	0	2	0	0.82	1
b8	++digit;	0	0	1	1	0	2	0	0.82	1
b9	else if (line.charAt(i) == ' ')	0	0	1	1	0	2	0	0.82	1
b10	++space;}}	0	0	1	1	0	2	0	0.82	1
	Execution Result (0=Successful, 1= Failed)	0	0	1	1	1				

scores. The suspiciousness score assigned to each basic block represents the possibility of that block being the root fault. In the example shown in Table IV below the basic blocks b7, b8, b9 and b10 have highest suspiciousness scores of 0.82 which is greater than the root fault b3.

Spectrum-based techniques use similarity metric to compute the suspiciousness of program entities. The similarity metrics of some well-known approaches are discussed in the section III-B. These metrics use program spectrum information derived from test case inputs to determine the correlation between program entities and test case results. The reasoning behind these techniques is that the program entities frequently executed by failed test cases are considered to be more suspicious. Thus, spectrum-based approaches compute the suspiciousness scores of program entities by analyzing the frequency in which these entities execute in failing and passing test cases.

In our example shown in Table IV, if we observe the execution trace of failed test case T3 we find that the fault context of b3 is {b1, b2, b5, b6, b7, b8, b9, b10} and the suspiciousness score of this fault context is defined as the sum of the suspiciousness scores of all basic blocks i.e. {b1, b2, b5, b6, b7, b8, b9, b10}. Similarly, for the program executed in failed test cases T4 and T5, the fault context of b3 is {b1, b2, b4, b5, b7, b8, b9, b10} and {{b1, b2, b4, b5, b6} respectively. The suspiciousness scores would be the sum of the suspiciousness scores of all basic blocks or statements of b3's fault contexts. So, there are three suspiciousness scores of b3's fault context. As we know that, a program entity's overall suspiciousness rank will be higher if the suspiciousness of that program entity is higher and the suspiciousness of its fault context is lower. Hence, we choose the minimum suspiciousness score of the three suspiciousness scores of b3's fault context.

The suspiciousness score of b3's fault context can be calculated as per the formula given below.

$S_c(b3) = \min [(S_b(b1)+ S_b(b2) + S_b(b5)+ S_b(b6)+ S_b(b7) + S_b(b8)+S_b(b9)+S_b(b10)), (S_b(b1) + S_b(b2) + S_b(b4)+ S_b(b5)+ S_b(b7) + S_b(b8) + S_b(b9)+S_b(b10)), (S_b(b1) + S_b(b2)+ S_b(b4)+ S_b(b5)+ S_b(b6))]$

$= \min (6.17, 6.26, 3.57)$

$=3.57$

Here, $S_c(b3)$ denotes the fault context suspiciousness score of basic block b3 and $S_b(b1)$, $S_b(b2)$, $S_b(b5)$ etc indicate the suspiciousness scores of b1, b2 and b5 respectively. Similarly, for instance, we can find out the suspiciousness score of b7's fault context as given below.

The program failure occurs when the faulty program entities are activated in an execution and the infected states are propagated through the program. Thus, the failure is dependent on the fault that is triggered and its context. By observing Table IV it is clear that b3 was activated in all test cases but failure is observed in T3, T4 and T5. This is because of the impact of b3's different contexts in different executions. In our example the root fault is b3 but it does have highest suspicious rank, instead b7, b8, b9 and b10 are ranked highest. Therefore, in order to improve the absolute fault rank of suspicious statements/ blocks it is required to combine the suspiciousness of program entities and their fault contexts to get to the final suspiciousness scores.

$S_c(b7) = \min [(S_b(b1)+ S_b(b2) + S_b(b3)+ S_b(b5)+ S_b(b6) + S_b(b8)+S_b(b9)+S_b(b10)),(S_b(b1) + S_b(b2) + S_b(b3)+ S_b(b4)+ S_b(b5) + S_b(b8) + S_b(b9)+S_b(b10)),(S_b(b1) + S_b(b2)+ S_b(b3)+ S_b(b4)+ S_b(b5)+ S_b(b6))]$

$= \min (6.13, 6.21, 4.34)$

$=4.34$

In our example basic blocks b7, b8, b9 and b10 are ranked higher than the root fault b3. These blocks were influenced by b3, and all their fault contexts include b3. So, the suspiciousness of b7's (or b8, b9, b10) fault context might have higher suspiciousness score as compared to b3's.

The following Table V summarizes the suspiciousness scores of all the basic blocks, suspiciousness scores of fault contexts of all basic blocks and the final overall rank of each basic block (i.e. program entity) based on the two suspiciousness ranks.

TABLE V
SUSPICIOUSNESS OF BASIC BLOCKS AND THEIR CONTEXTS

Basic Block No	Ochiai		Fault Context		Ochiai with Fault Context	
	S_b	R_b	S_c	R_c	R_b+R_c	Rank
b1	0.77	5	0.36	1	6	1
b2	0.77	5	0.36	1	6	1
b3	0.77	5	0.36	1	6	1
b4	0.67	9	0.37	5	14	9
b5	0.77	5	0.36	1	6	1
b6	0.58	10	0.38	6	16	10
b7	0.82	1	0.43	7	8	5
b8	0.82	1	0.43	7	8	5
b9	0.82	1	0.43	7	8	5
b10	0.82	1	0.43	7	8	5

c. *Framework*

The main process of this fault context based approach is shown in Fig. 2. This approach mainly consists of the following three steps.

i. The program under test is executed with test suites containing failed and passed test cases, and program spectrum data is collected.
ii. Then using one of the existing classic SBFL techniques (Ochiai in our study) suspiciousness scores are computed for program statements and their fault contexts.
iii. Based on the two suspiciousness scores, the final improved suspiciousness rank list of program statements is generated in the form of a debugging report.

Now we formally describe the steps of the fault context based approach of fault localization [13].

Let P={e_1, e_2, e_3, …,e_y} be a faulty program which contains program entities e_i. Let T = {t_1, t_2, …, t_n} be a test suite containing t_i test cases. This test suite T can be divided into two subsets T_p and T_f which represent passed and failed test cases respectively. As per our example we have T_p = {t1, t2} and T_f = {t3, t4, t5}. Similar to existing spectrum-based approaches we first collect the program spectra by executing the program under test P with the input of test cases T_p and T_f. We then compute and analyses the suspiciousness scores of each program entity and their fault contexts. Finally, a ranking list in descending order for P is generated that shows the likelihood of each program entity to be faulty. This approach has three major steps as explained below.

Computation of suspiciousness for program entities

In this step we collect program spectra or coverage matrix M, by executing the program P with the input of test suite T. The result vector r is collected which contains the results data as per the execution of passing (T_p) and failing (T_f) test cases. We then compute the suspiciousness score for every program entity by using a spectrum-based similarity metric. In our case we use Ochiai metric as given in (2) of section III - B.

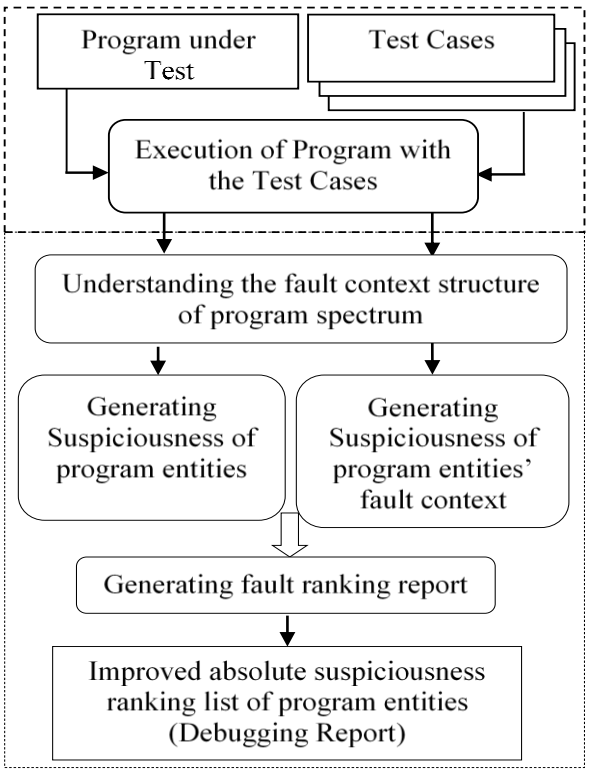

Fig. 2. Framework for fault context based fault localization method

Computation of suspiciousness for fault context of program entities

In this step we formally define the fault context. For a failed execution i, the covered set of program entities are represented as $ec_i = \{c_1,\ldots, c_j, \ldots, c_k\}$. The fault context of c_j is the set of all program entities covered by the failed execution of t_i except the entity c_j itself. The fault context of c_j can be denoted as follows.

$$\text{faultcontext}_c(c_j, t_i) = ec_i / c_j \quad (6)$$

The c_j's suspiciousness score can be calculated as the summation of all suspiciousness scores of program entities ec_i in test case execution t_i except for the entity c_j.

$$S_c(c_j, t_i) = \sum S_b(ec_i(k)) \quad (7)$$

In case if there are more than one fault contexts for the program entity c_j in T_f, then we have to find the minimum of these fault contexts for c_j. The fault context of c_j can be defined as.

$$\text{faultcontext}_c(c_j) = \{ec_i / c_j \mid i \in \text{set of failed test executions}\} \quad (8)$$

$$S_c(c_j) = \min(S_b(e_j, t_i) \mid i \in \text{set of failed test executions}) \quad (9)$$

Fault ranking list generation

After the steps 1 and 2, two fault rank list can be generated, first for the suspiciousness scores of program entities, and second for their fault contexts. In this step we further generate a new third rank list.

Assume S is a program entity then the rank of S will be higher in the new ranking list, if the suspiciousness score S_b (S) is higher and the suspiciousness score of its fault context S_c (S) is lower.

To create a new fault ranking list R we first create two fault ranking lists for a faulty program, R_b and R_c where, R_b is in descending order of S_b and R_c is in ascending order of S_c. The two ranking lists are combined to generate the new ranking list R as follows. Assume e_i and e_j are two suspicious program entities and e_i has ranks R_b^i and R_c^i , and e_j has ranks R_b^j and R_c^j . In the new fault ranking list R, ei is ranked higher than ej if and only if $R_b^i + R_c^i <= R_b^j + R_c^j$.

By observing Table IV we can see that with traditional method of fault localization it takes 7 searches to reach to the faulty statement S3. Whereas with the new fault context based approach we need to search only 3 statements to reach to the faulty statement. That means the developer's effort will be reduced by 57.14% because of improvement in absolute rank of suspicious program entity.

D.Statistics Based Techniques

Dynamic fault localization techniques do not make use of prior knowledge of semantics of programs under test. These techniques however, identifies whether an execution is successful or failed. So, dynamic techniques locate program bugs by differentiating unsuccessful and successful execution runs. The techniques based on predicate evaluations are promising methods of fault localization [14].

Program run time behavior is characterized in the form of statistics such as evaluations of conditionals and function return values. For example, the predicate "i < STRLEN" where the variable i is an index and it is being checked whether or not it is exceeding the max length of the buffer. In this way, the statistics of multiple executions at run time are collected and recorded for later analysis. A statistical debugging method known as LIBLIT05 [14] is discussed here for a reference. For each conditional or predicate P in a program Pr, LIBLIT05 computes two conditional probabilities as given below.

Prob1= Prob (Pr fails | P is ever observed)

and

Prob2= Prob (Pr fails | P is ever observed as true)

Then the difference of probabilities Prob2 – Prob1, indicates the relevance of P to the fault. That means, a predicate is related to a fault, if there is a correlation between its true evaluation and the program failure. We can also say that those predicates that have the above difference Prob2- Prob1 <=0, can be discarded. The importance scores of remaining predicates are calculated and these predicates are prioritized as per their scores. These important scores indicate the relationship between predicates and program faults. Those predicates which have higher scores are examined first.

Another statistical model based approach was known as SOBER which was proposed by Liu et al. [14]. This fault localization technique does not use any prior knowledge of program semantics. This technique ranks suspicious predicates. The evaluation patterns of predicates consider both successful and unsuccessful executions in SOBER technique. This method considers a predicate as faulty, if its evaluation pattern in unsuccessful executions differs considerably from the successful executions. When a test case is executed, it is possible that a predicate P is evaluated as true more than once. The following formula of (10) gives the probability that the predicate P is evaluated true in each execution of a test case.

$$\pi\,(P) = \frac{n(t)}{n(t)+n(f)} \tag{10}$$

Where n(t) and n(f) are number of times P is evaluated as true and false respectively. Now, the distribution of π (P) in successful and failed executions of test cases is checked. If there is a significant difference in the distribution of π (P) in failed and successful executions then P is considered as faulty.

Cross tabulation is another statistical analysis based technique for fault localization that calculates suspiciousness of program statements [15]. This technique utilizes information related to execution results i.e. success or failure and statement coverage information with respect to different test cases. The structure of crosstab is such that it has two columns that specify two categorical variables – covered and not covered; and it has two row wise categorical variables for successful and failed executions. The dependence or independence between coverage of each statement and execution results is determined using a hypothesis test. The degree of association between the execution results and coverage of each statement is measured using chi square statistic test. So, the suspiciousness of each statement depends on this degree of association.

So, it is important to note that the methods like SOBER and LIBLIT05 are used to only rank predicates which are likely to cause errors whereas crosstab method is used to find suspicious program elements like statements, functions, predicate etc.

A predicate with two or more conditions is called a short circuit evaluation and it may occur often in program execution. In this short circuit evaluation if first condition is suffice to evaluate the result of the predicate then, the rest of the conditions that follow will not be executed or evaluated. So, short circuit evaluations of individual predicates can be identified and a set of evaluation sequences for each predicate is generated. The "debugging through evaluation sequence approach (DES)" uses such information and can be compared with predicate based approaches like SOBER and Liblit05.

Another statistical method which uses the behavior of two sequentially connected predicates in the execution was studied by You et al. [16]. For each execution of a test case a weighted execution graph was constructed where nodes of the graph represent predicates and edges denote transition of two sequential predicates. A suspiciousness value is computed for each edge to measure the possibility of its fault proneness.

E. Program State-Based Techniques

Variables and their values at run time make program state which can be an indicator for bug localization. One of the approaches of program state based technique is to change the values of some of the variables to find out which one causes erroneous program execution. Delta debugging was suggested by Zeller and Hildebrandt [17] in which the differences in program states are calculated between executions of a successful test and a failed test through their memory graphs. To test suspiciousness of variables, a program is tested with successful test and values of variables are replaced with related values from the same place in a failed test, and program execution is repeated. A variable is considered as suspicious when a same failure is observed. The delta tool is very popular in software industry and is being used extensively for automated debugging. Cause transition technique is an extension of delta debugging which was proposed by Cleve and Zeller [18]. In this technique, when the cause of failure changes from one variable to another, such locations and times are identified. To detect cause transitions in a program execution the algorithm named as CTS was used.

It is apparent that, program executions may consist of thousands of states and each matching point requires additional test executions by delta debugging to narrow down the causes. So, the cause transition technique is comparatively a high cost approach. It is also not necessary that the locations identified by this technique may not be the places where the fault exists. To overcome this issue, the cause transition was extended to "failure-inducing chop" by Gupta et al. [19]. In the first step, input output variables that are causes of failure are identified by delta debugging method. Dynamic slices are then constructed for these variables. Now the code is considered as suspicious which is at the intersection of forward and backward slicing of the input and output variables respectively. The delta debugging

still has some limitations like it is difficult to handle confusing partial state replacement, errors caused by omission of execution and efficiency of delta debugging is poor. Later, this limitation was addressed by a cause inference model that explains the difference between a failed execution and a successful execution.

A value profile based technique presented by Jeffrey et al. [20] helps developer in software debugging. In this method interesting value mapping pairs (IVMPs) are figured out in program statements and these values are changed so that correct output can be produced by failed test cases. Different test cases are executed to produce profiling information which is used to generate alternative sets of values. For each statement instances of every failed test case, different alternative value sets are used to perform value replacements. Then, each statement is given a rank on the basis of these IVMPs as per the number of failed executions where at least one interesting value mapping pair is identified for that statement. This statement ranking helps identifying the location of fault.

Zhang et al. [21] studied that a fault within a statement may spread a sequence of infected program states before the failure is noticed. They also said that, a particular program statement which is executed by a series of failed test cases might not be the root cause of the failure. This can be explained by an example, suppose that a statement T on a branch B has the function of setting up a null pointer variable. It is further assumed that this pointer variable will not be used to execute any function, until a different distant (in context of data or control dependence) statement T' on a branch B' has been accessed, which will cause the program failure. If T is exercised in many executions through various test cases that do not reveal any failure, the statement T or its directly related branches cannot be really identified as suspicious. The coverage based techniques like Tarantula will give higher suspiciousness rank to T' than T, in case of above explained scenario. If data flow analysis is employed in this scenario then it will show the usage of null pointer and help estimating the suspiciousness of T, T', B and B'. It is also important to note that data flow profiling is expensive.

Zhang et al. [21] described the concept of edge profiles which represent successful and failed executions. A given program is abstracted as a CFG (control flow graph) and sampled a program execution as an edge profile. These edge profiles tell which edges of the CFG are traversed during the execution and quantify changes in program states over an edge according to the different test case executions of the edge. This dissimilarity between edge profiles is used to model how each basic block may cause failures by abstractly spreading infected program states to its neighboring basic blocks through control flow edges. In this way, Zhang et al. measured suspiciousness of infected program states propagated through each edge, related the basic blocks with edges through such propagation of infected program states, estimated suspiciousness value of each basic block, and demonstrated a ranked list of program statements which helped finally in identification of faults.

F. Machine Learning-Based Techniques

Machine learning based techniques can be applied to identify or learn the location of fault on the basis of statement coverage data as an input and result of execution of test cases i.e. success or failure.

A fault localization technique was given by Wong and Qi [22] which used back-propagation neural network. The neural network is trained by the coverage data of test cases and their equivalent execution results which help neural network to understand the relationship between them. Then, the trained neural network is inputted with test cased that each covers only a single statement of the program and the probability of the faulty statement is outputted. This back-propagation neural network was also later extended for object oriented programs by researchers.

The C4.5 decision tree algorithm classifies test cases into different partitions so that failed test cases can be identified. The basis is that the different failure conditions for test cases can be identified based on the input and output of test cases. This is called category partitioning. The failure conditions which originate from different faults are represented by different paths in the decision tree. These decision tree paths represent rules that model different failure conditions that ultimately give different failure probability predictions.

G.Data Mining-Based Techniques

Data mining techniques, which work on the similar lines of machine learning, construct a model using relevant information extracted from data. Data mining can be used in fault localization for example we can identify the pattern of execution of statements in a program that leads to a failure. Due to the huge volume of data the complete execution traces of a program cannot be analyzed manually, so data mining techniques can be wisely applied to execution traces.

Statement sub-sequences of length N from trace data, is known as N-grams. The N-grams occurring higher than a pre specified limit are searched by examining the failed execution traces. The confidence for a particular N-gram is determined by computing the conditional probability that a particular N-gram occurs in a given failed execution trace. This N-gram analysis is used to rank suspicious statements in a program by arranging the N-grams in descending order of confidence along with the corresponding statements in the program.

Fault localization based on association rule analysis was discussed by Cellier et al. [23]. This method attempts to find out rules concerning the association between coverage of statements and corresponding execution failures. The occurrence of this association rule is computed. A threshold is chosen to specify that a selected rule is required to cover a minimum number of failed executions. Faults are located by examining the rankings of these generated rules.

H.Model-Based Techniques

In model based analysis of programs, models serve as oracles of programs being under analysis. The behavior of actual program and behaviors of models are compared in order to find out bugs in the programs. Whereas, in model based fault localization, models may contain bug when generated directly from actual programs. The expected results of programs provided by testing engineers or

programmers are compared with observed execution behaviors and differences are used to identify components of models due to which such misbehaviors are observed.

Static or dynamic analysis is performed to find dependencies between statements in a program and thus dependency based models are generated. A functional dependency model was described by Mateis et al. [24] which used dependency based models to explain the structure of a program. Whereas, the logic based languages such as first order logic was applied to model behavior of the target program. They presented this functional dependency model for JAVA programming language features such as class, methods, assignment and conditional statements, loop constructs etc. The unstructured control flows in JAVA programs like recursive method calls, jump and return statements, exceptions are handled by extended dependency based models.

The dependence graph concept was also broadened to model program behavior over a group of test cases. The probabilistic program dependence graph was used to model reasoning about uncertain program behaviors that are possibly associated with program bugs.

According to Wotawa et al. [25], program structure and behavior can be represented by dependency based models constructed by first order logic after analyzing the source code. The first order logic is also used to represent test cases with their expected outputs. Now, the target program under analysis is run with the test cases and if it fails then conflicts with the models and test cases are used to detect doubtful statements that cause the failure. The constraint on this study is that it only focuses on programs which do not have loop constructs. To deal with this limitation, Mayer and Stumptner [26] worked on abstraction based model which used abstract explanation to handle loops, heap data structures and recursions.

Value based models are also used to locate bugs in programs by representing data flow information in programs. These models are appropriate for small programs as they are more computationally intensive.

We now give an introduction to model checking based methods for fault localization that use model checkers to locate bugs [27].

If a program does not work as per its requirements, or specifications a model checker can help by providing a counterexample which is a run of the program from the start of the program to the point where failure is noticeable. The programmer can trace the program line by line using a debugger to check the places in the counterexample to visualize the data. But when a program with hundreds or thousands line of code is to be debugged it very difficult and tedious to locate the bugs.

Ball et al. [28] suggested that a model checker can be used to investigate all the paths of a program apart from that of the counter example. The execution paths that do not cause a failure that means the successful paths are noted down. Algorithms were used to identify the transitions that are found in the execution path of the counter example but not in the execution paths which executed successfully. The possible causes of bugs are those components of the program related to these transitions. This drawback of this method is that it is very expensive to compute all successful execution paths.

Program executions can be represented by variable assignments. The distance between two program executions can be measured by a distance metric defined by Groce et al. [29]. A successful execution was produced which is closest to counter example by a model checker. Now the difference between successful execution and the counter example was computed which provided the probable location of the bug with an explanation.

IV. Evaluation Metrics

We know that the bugs in programs may exist non-contiguously and across the multiple modules so the inspection of the suspicious code stops as soon as we find the first bug. The focus of fault localization is to help developers and coders to find the starting point of fault so that the error fixing can be initiated rather than to locate the each and every piece of code that needs to be added, changed or deleted related to the each bug. Because of this reason, we can measure the effectiveness of software fault localization by computing the percentage of the code that needs to be examined before locating the starting point of a bug. Here by code we mean statements, predicates, functions etc.

T-score as defined below in (11) computes the fraction or percentage of code that we need to examine to reach to the place of the fault [14]. We explain this measure as follows.

Assume a program dependence graph (G) is given for a faulty program (P). The graph G consists of vertices which represent program statements and the edges between them represent data and/ or control dependencies. The statements which contain faults are indicated by defect vertices, and V_{defect} denotes a set of defect vertices. A fault localization report R, consists of a set of suspicious statements and they are represented as V_{blamed}. A developer starts debugging from V_{blamed} and performs breadth first search until he finds any defect vertices. The set of statements traversed during this search is written as $V_{examined}$.

$$T = \frac{|Vexamined|}{|V|} * 100\% \quad (11)$$

Here, |V| is the magnitude of the PDG (program dependence graph). Some authors use 1 - T as an equivalent measure of T-Score. When a fault localization report is provided, the T-score estimates the percentage of code that a programmer requires to examine before the location of fault is found. A small set of statements that contain location of fault is considered to be high quality fault localization.

Jones and Harrold [5] used a different type of T-score to present the localization results of Tarantula. Tarantula gives a ranking of all executable statements and the T-score can be calculated directly by examining this ranking. A programmer can examine statements one by one from top of the ranking until a faulty statement is found. The T-score can be calculated as percentage of statements examined. This method is called ranking based T-score as against the PDG based T-score.

The EXAM score or Expense score [5], [22] given in (12) is another measure for calculating the percentage of code required to be checked before reaching the location of the first faulty statement.

$$\text{Exam Score} = \frac{\text{Number of statements examined}}{\text{Total number of statements in the program}} \quad (12)$$

In [5], authors consider executable statements in place of total count of statements. A technique explained by Liu et al. [30] uses EXAM score to find percentage of predicates (not statements) that need to be checked. The predicates are arranged in descending order of their relevance to faults. The P-score also uses the same method as given below in (13).

$$\text{P} - \text{Score} = \frac{1-\text{base index of P in S}}{\text{count of predicates in S}} \text{ X } 100\% \quad (13)$$

Where S is the list of predicates arranged in a logical order (sorted), P is predicate that is most relevant to a fault and the 1-based index specifies the first predicate of S is indexed by 1 (not 0).

We can observe that the lower the EXAM score, the fault localization is more effective whereas, the lower the T-score the technique is less helpful.

A variation of EXAM score called total developer expense (D) is used to measure the total effort to locate faults in a program which contains multiple faults. It is the sum of the EXAM scores for all faults in a program.

Wilcoxon signed rank test [31] is another metric based on statistical approach in which if we assume that there are two techniques P and Q and P is more effective than Q. We inspect an alternative hypothesis that Q requires inspecting an equal or more number of statements than P in order to find out the location of a bug. If the alternative hypothesis is accepted with a certain confidence then it leads to ascertain that whether P is statistically more helpful in finding out the fault. Another metric that provides a global view is the total count of statements that needs to be inspected to find out all faults in a given situation.

According to recent user studies, developers tend to investigate only the top 5 or top 10 elements in the recommendation list provided by fault localization methods before giving up the debugging process. The concept of accuracy, as defined by Sohn and Yoo [32], can be used to compare the efficiency of two fault localization approaches. It counts the number of faults that have been localized within the ranking's Top-N positions.

It is essential to understand that the effectiveness of software fault localization techniques depend on other factors also such as computational cost, time and space complexity in terms of data collection, human efforts and tool support. Besides that, human factors also need to be considered such as debugging process of developers, how cause-effect chains of failures are revealed by them, how they work upon the solutions beyond a suspiciousness ranking of program code. No study has been reported that incorporates all these factors in an evaluation method.

V. Experimental Study

In section III-C we presented two techniques that improve the effectiveness of existing spectrum-based fault localization techniques in single fault context. The first one was based on failed execution slice and the second technique utilizes the concept of fault context to improve the absolute rank of faulty program entities. In this section we present the experimental work conducted to validate the effectiveness of the two techniques. We use standard benchmark Siemens programs for experimental work. The Siemens suite contains seven programs and each one has a correct version and a set of faulty versions of the same program.

A. Experimental Design

We have conducted two sets of experiments in support of the two methods which improve the performance of SBFL and were illustrated with the help of working examples in section III-C. The first method uses the concept of failed execution slice (FES) and the second method combines the suspiciousness of program entity and suspiciousness of program entity's fault context to improve the absolute rank of faulty entities in the program under test. The aims of the experiments are to show the improvements in performance of existing SBFL techniques using the two methods. We use Ochiai SBFL metric to compute the suspiciousness calculation because Ochiai is an efficient technique and has been often referred in the fault localization literature [33], [34]. The formula for the calculation of suspiciousness score using Ochiai method is given in (2) of section III.

The experimental study is performed on standard benchmark (Siemens suite) [http://sir.unl.edu/portal/index.php]. It contains seven programs: print_tokens, print_tokens2, replace, schedule, schedule2, tcas and tot_info, each of which has a set of faulty versions [35], [36]. The details of the subject programs are given in Table VI.

Table VI
Subject programs used in empirical study

Program	LOC	Faulty Versions	Test Cases	Brief Description
print_tokens	565	7	4130	Lexical analyzer
print_tokens2	510	10	4115	Lexical analyzer
schedule	412	9	2650	Priority scheduler
schedule2	307	10	2710	Priority scheduler
tcas	173	41	1608	Altitude separation
tot_info	406	23	1052	Information measure
replace	562	32	5542	Pattern recognition

Our experimentation work uses GCOV (GCC) 10.2.0 [https://gcc.gnu.org/onlinedocs/gcc/Gcov.html] on Linux platform to collect the coverage data. We used different faulty versions of subject programs in our experimentation work. Table VII shows the details of the faulty versions used in the experimentation work performed on two methods – FES based and fault context based methods. We used 21 faulty versions of Siemens programs in our experimentation. To compute the suspiciousness score and other results in our experimentation work we developed an automated tool in Python 3.7.3. All experiments were performed on Linux environment running on top of Windows 10 machine with Intel® Core i5 CPU 2.7 GHz and 8 GB of RAM.

Table VII
Faulty Siemens Versions Used in Experimentation

Program	FES	Fault Context
print_tokens	V5	V5, V7
print_tokens2	V5	V5, V7
schedule	V3	V2, V3
schedule2	V6	V5,V6
tcas	V2	V2, V5
tot_info	V5	V5,V9
replace	V8	V3, V8

B. Evaluation Metrics

To evaluate the effectiveness of fault localization we use three evaluation metrics, namely EXAM score, Top-N and Wilcoxon signed-rank test.

i. *EXAM score*

As given in (12) of section IV, the EXAM score is defined as the percentage of statements that need to be examined in a program before reaching to the first faulty statement. A lower expense score indicates a better performance.

We evaluate the fault localization performance using EXAM score metric with absolute and relative variants. The absolute metric is defined as the number of statements that need to be examined in a program before reaching to the first faulty statement and relative EXAM score is defined as the percentage of statements that need to be examined in a program before reaching to the first faulty statement. So, the relative version of EXAM score metric is compared to the length of the rank list, which the program size.

The two approaches presented in section III C (FES based and Fault Context based) improve the performance of classical SBFL methods. These two approaches improve the absolute rank of faulty program statement in the fault rank list. Therefore, the effectiveness of these new approaches can be measured in terms of improvement when we compare it with the classic SBFL metric. The improvement formula can be defined as.

$$\text{Improvement}\ (\mathrm{P},\mathrm{Q}) = \frac{P-Q}{P} \times 100\% \quad (14)$$

Where P is the absolute rank generated by the classic SBFL method and Q is the absolute rank given by the new approaches i.e. FES based and Fault Context based methods.

ii. Top-N

Top-N indicates the number of faults a fault localization technique ranks among the top-N (N=1, 5 or 10 and so on) positions in the ranked list. The metric would be stricter if we have the smaller value of N. For example, Top-5 means all faults are raked within top 5 positions in the ranked list. Top-N is a frequently used metric in the fault localization literature. In our study we use top-N metric to evaluate the effectiveness of FES and fault context based methods in comparison to traditional Ochiai method.

iii. Wilcoxon Signed-Rank Test

Wilcoxon signed-rank test is an alternative option to other existing hypothesis tests such as z-test and paired student's t-test particularly when a normal distribution of a given population sample cannot be assumed [37].

Wilcoxon signed-rank test is also utilized to give a comparison with a concrete statistical basis between two or more techniques in terms of effectiveness. In this paper our aim is to experimentally show that the two fault localization techniques i.e. FES based and Fault Context based are more effective as compared to the traditional Ochiai method in most of the cases. We compare the performance using the EXAM score metric which computes the total number of statements that a developer needs to check on all techniques before identifying the first faulty statement. Therefore, an evaluation will be conducted on the one-tailed alternative hypothesis that the other technique (in our case Ochiai) used for cross-comparison require the examination of an equal or greater number of statements than the two techniques (FES based and Fault Context based). The null hypothesis is stated as follows:

H0. The number of statements examined by the traditional Ochiai technique is less than or equal to the number of statements examined by the FES based and Fault Context based techniques.

Therefore, if H0 is rejected, the alternative hypothesis is accepted. The alternative hypothesis implies that the both FES based and Fault Context based techniques will require the examination of lesser number of statements than the traditional Ochiai technique which indicates that the FES based and Fault Context based techniques are more effective.

C. Results and Discussions

In this section, we present our detailed experimental results on two approaches i. e. FES based and Fault Context based methods described in section III C that improve the performance of spectrum-based (lightweight) fault localization.

As described in subsection C of section V we have used three evaluation metrics (namely EXAM score, Top-N and Wilcoxon signed-rank test) for the assessment of the experimental results. We present the experimental results based on EXAM score and Top-N metrics separately for the FES based and Fault Context based methods in the following two subsections. The evaluation based on Wilcoxon signed-rank test is presented together for the two methods in the end of the results and discussions section.

Experimental Results on FES based method

In this subsection we compare the effectiveness of FES based method with classic Ochiai method. The experiments were conducted on 7 faulty versions of Siemens programs as shown in Table VII.

Table VIII shows the improvement in fault localization performance using Failed Execution Slice (FES) based method. First three columns of Table VIII show the Siemens program name, faulty version and line of code (LOC) respectively. The performance of fault localization using classic SBFL method (Ochiai metric in our study) is shown in columns 4 and 5, where column 4 shows the EXAM score in terms of the absolute measure and column 5 shows the EXAM score in terms of relative measure. Similarly, the performance of fault localization using FES based method is shown in columns 6 and 7 respectively.

We take an example of print_tokens program to compare the effectiveness of FES based method against the classical SBFL method (i. e. Ochiai). It can be observed that it takes 9 searches to reach to the faulty statement, whereas FES method requires only 7 searches to locate the fault. If we analyze in terms of relative expense score, we can see that with FES based method we need to examine 1.24% of statements (of total LOC of print_tokens program) as against 1.59% statements when we use classical SBFL method. Therefore, we can say there is an improvement of 22.22% when using FES based method as against classical SBFL method. In other words, developer's effort has been reduced by 22.22% as shown in column 10 of Table VIII.

TABLE VIII
IMPROVEMENT IN FAULT LOCALIZATION PERFORMANCE WITH FES BASED METHOD

Subject Program	Version	LOC	Code Examined				Difference		Improvement %
			Using Classic Ochiai Method	EXAM Score %	Using FES Method	EXAM Score %	Code Examined	EXAM Score %	
print_tokens	V7	565	9	1.59	7	1.24	-2.00	-0.35	22.22
print_tokens2	V5	510	32	6.27	25	4.90	-7.00	-1.37	21.88
schedule	V3	412	28	6.80	27	6.55	-1.00	-0.24	03.57
schedule2	V6	307	23	7.49	11	3.58	-12.00	-3.91	52.17
tcas	V2	173	11	6.36	10	5.78	-1.00	-0.58	09.09
tot_info	V5	406	6	1.48	4	0.99	-2.00	-0.49	33.33
replace	V8	562	17	3.02	9	1.60	-8.00	-1.42	47.06
Average			18.00	4.72	13.29	3.52	-4.71	-1.20	27.05

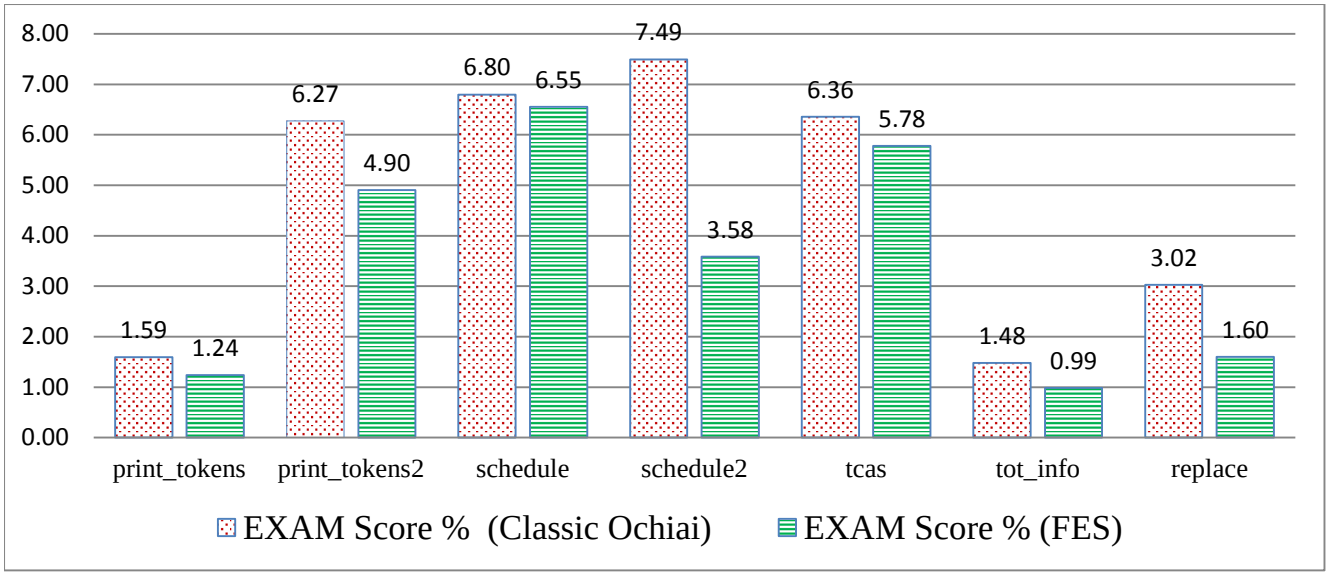

Fig. 3. Comparison of Fault Localization Performance between classic Ochiai and FES based method in terms of EXAM score

The last row of Table VIII summarizes the overall fault localization performance with the corresponding average values of each column. We can see there is an overall improvement of 27.05% in fault localization performance when FES based method is used.

The comparison of fault localization performance between traditional Ochiai and FES based methods is also shown graphically in Fig. 3. The vertical axis represents the EXAM scores, which means the percentage of code need to be searched before locating the faulty statement.

Table-IX shows results comparing fault localization performance between traditional Ochiai and FES based methods in terms of Top-N metric. It can be observed that, with the Ochai method we are unable to locate any faults among Top-1 and Top-5 positions, but we can find 28.57% of faults (i. e. 2 out of total 7 faults) by examining the Top-10 positions (i. e. statements) in the suspiciousness ranking list. In contrast, if we see the results with FES based method, we are able to locate 14.29% of faults among the Top-5 positions and 42.86% of faults (i. e. 3 out of total 7 faults) can be located by examining the Top-10 positions in the suspiciousness ranking list. Therefore, the results show that the FES based method is giving better fault localization performance as compared to the traditional Ochiai method.

TABLE IX
Percentage of faults successfully located at each Top-N metric by Ochiai and FES based method

Technique	Top-1	Top-5	Top-10
Traditional Ochiai	0	0	28.57
FES based method	0	14.29	42.86

TABLE X
IMPROVEMENT IN FAULT LOCALIZATION PERFORMANCE WITH FAULT CONTEXT BASED METHOD (SET-1)

Subject Program	Version	LOC	Code Examined				Difference		Improvement %
			Using Classic Ochiai Method	EXAM Score %	Using Fault Context Method	EXAM Score %	Code Examined	EXAM Score %	
print_tokens	V5	565	17	3.01	15	2.65	-2.00	-0.35	11.76
print_tokens2	V5	510	13	2.55	9	1.76	-4.00	-0.78	30.77
schedule	V2	412	50	12.14	15	3.64	-35.00	-8.50	70.00
schedule2	V5	307	10	3.26	6	1.95	-4.00	-1.30	40.00
tcas	V2	173	5	2.89	4	2.31	-1.00	-0.58	20.00
tot_info	V5	406	6	1.48	4	0.99	-2.00	-0.49	33.33
replace	V3	562	51	9.07	14	2.49	-37.00	-6.58	72.55
Average			21.71	4.91	9.57	2.26	-12.14	-2.66	39.77

TABLE XI
IMPROVEMENT IN FAULT LOCALIZATION PERFORMANCE WITH FAULT CONTEXT BASED METHOD (SET-2)

Subject Program	Version	LOC	Code Examined				Difference		Improvement %
			Using Classic Ochiai Method	EXAM Score %	Using Fault Context Method	EXAM Score %	Code Examined	EXAM Score %	
print_tokens	V7	565	9	1.59	7	1.24	-2.00	-0.35	22.22
print_tokens2	V7	510	14	2.75	9	1.76	-5.00	-0.98	35.71
schedule	V3	412	24	5.83	23	5.58	-1.00	-0.24	4.17
schedule2	V6	307	23	7.49	11	3.58	-12.00	-3.91	52.17
tcas	V5	173	20	11.56	5	2.89	-15.00	-8.67	75.00
tot_info	V9	406	15	3.69	8	1.97	-7.00	-1.72	46.67
replace	V8	562	45	8.01	33	5.87	-12.00	-2.14	26.67
Average			21.43	5.85	13.71	3.27	-7.71	-2.57	37.52

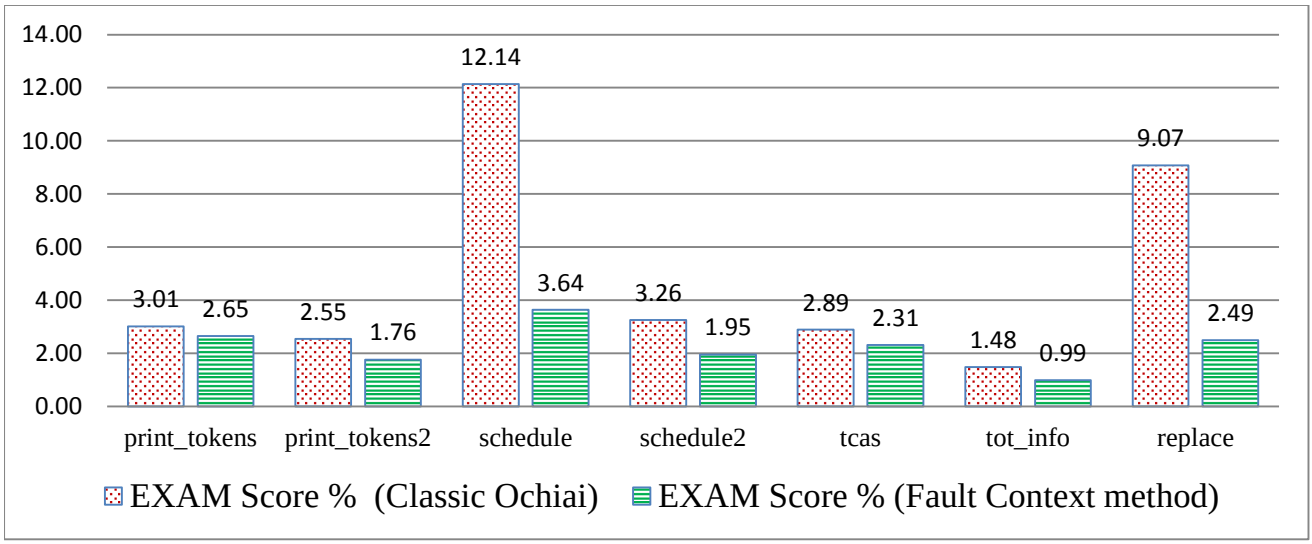

Fig. 4. Comparison of Fault Localization Performance between classic Ochiai and Fault Context based methods in terms of EXAM score (Set-1)

The fault localization results using Top-N metric shown in Table-IX can also be explained in other words with the help of Table-VIII. In case of tot_info and print_tokens subject programs, we need to examine 6 and 9 statements (see column 4) respectively in order to locate the faults. That means the faults can be located by examining Top-10 statements in the suspiciousness rank list when we use traditional Ochiai method.

Whereas, when we use FES based method, we need to check 4 statements (see column 6) in case of tot_info program to locate the fault, which means we are checking Top-5 statements in the ranking list. Similarly, in case of print_tokens, replace and tcas subject programs we need to check 7, 9 and 10 statements respectively to identify the faulty statements, which means we are examining Top-10 statements in the ranking list to locate the faults. For the remaining 3 subject programs (i. e. schedule2, print_tokens2 and schedule) we need to examine more than 10 statements to locate the faults as shown in Table-VIII. This can be termed as Top-all.

Experimental Results on Fault Context based method

In this subsection we compare the effectiveness of fault context based method with classic Ochiai method. We have conducted two sets of experiments on 14 faulty versions of Siemens programs as shown in Table VII.

Table X and Table XI show improvements in fault localization performance achieved by Fault Context based method as against classic Ochiai method on different faulty versions of Siemens programs.

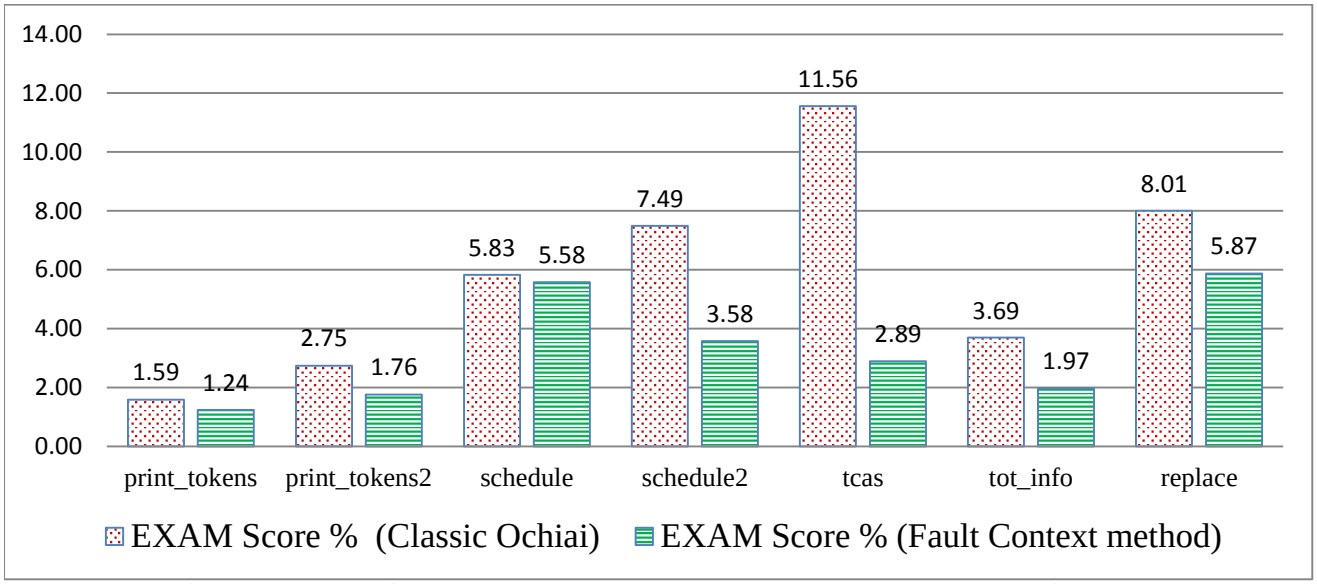

Fig. 5. Comparison of Fault Localization Performance between classic Ochiai and Fault Context based methods in terms of EXAM score (Set-2)

The columns 4 and 5 show the fault localization performance in terms of absolute rank and EXAM score percentage when we use classic Ochiai method. Similarly, the columns 6 and 7 show the performance with respect to Fault Context method. The column 10 shows the improvement achieved by Fault Context methods in comparison to classic Ochiai method. For example, in case of print_tokens program in Table X we need to go through 17 statements to locate the actual faulty statement and with fault context method we need to search only 15 statements to locate the fault. Therefore, we observe an improvement of 11.76%. This means developer needs to search 11.76% less statements in order to reach to the faulty statement.

In the same way when we observe print_tokens program of Table XI, there is an improvement of 22.22% in fault localization performance when we use fault context method as compared to the classic Ochiai method.

The average improvement is shown in the last row of Table X and Table XI. We observe an average improvement of 39.77% in the first set of experimental results (Set-1) shown in Table X, and in case of second set of experimental results (Set-2) shown in Table XI has an average improvement of 37.52%. We see a significant improvement in fault localization performance when fault context method is used as against the classic Ochiai method.

Therefore, it can be concluded that both FES based and fault context based methods further improves the fault localization performance.

For better readability, the comparisons of fault localization performance between traditional Ochiai and fault context based method are also shown graphically in Fig. 4 and Fig. 5 for the two sets of experiments respectively. Both sets of experiments have been carried out in same experimental settings but with different faulty versions of Siemens programs.

Now we compare the fault localization performance between Ochiai and fault context based method in terms of Top-N metric.

Table-XII provides a comparison of fault localization results between Ochiai and fault context based methods using Top-N metric. When we use traditional Ochiai method, we are unable to find any fault in the top-1 position, but 7.14% (1 out of total 14 faults) of faults and 21.43% of faults (i. e. 3 out of total 14 faults) can be found among Top-5 and Top-10 positions respectively. However, the fault context based method is also unable to locate the fault at Top-1 position, but it can locate 21.43% of faults (3 out of total 14 faults) and 35.71% of faults (i. e. 5 out of total 14 faults) among Top-5 and Top-10 positions respectively. Based on the results shown in Table-XII, it can be clearly observed that the fault context based method provides better fault localization performance over traditional Ochiai method.

To make the experimental conclusions more substantial, we now present the evaluation approach with respect to Wilcoxon signed-rank test. The Wilcoxon signed-rank test provides a reliable statistical basis for comparing the effectiveness of different techniques and has been widely used in earlier fault localization studies.

TABLE XII
Percentage of faults successfully located at each Top-N metric by Ochiai and Fault Context based method

Technique	Top-1	Top-5	Top-10
Traditional Ochiai	0	7.14	21.43
Fault Context based method	0	21.43	35.71

TABLE XIII
The confidence with which it can be asserted that a FES-based approach is more effective than Ochiai

	Subject programs of Table-VIII
Ochiai	99%

TABLE XIV
The confidence with which it can be asserted that a Fault Context based approach is more effective than Ochiai

	Subject programs of Table-X	Subject programs of Table-XI
Ochiai	99%	99%

Table XIII and Table XIV show the effectiveness comparison of FES based and Fault Context based methods respectively, as against the traditional Ochiai method using the Wilcoxon signed-rank test. The tables give the confidence of which the alternative hypothesis can be accepted (that the two approaches FES and Fault Context based methods require the examination of fewer statements than the compared baseline approach that is Ochiai). One can observe that with 99% confidence (with a significance level of 0.01), FES and Fault Context based methods are more efficient than the traditional Ochiai method on all faulty versions. In other words, we find that the EXAM scores of FES and Fault Context based methods are statistically better than that of the traditional Ochiai method.

Therefore, the results from the Wilcoxon signed-rank test evidently show that the FES based and Fault Context based methods are more effective than the classic Ochiai on Siemens programs, and can further improve the performance of existing SBFL methods such as Ochiai. The results are also consistent with our previous conclusion that the two methods perform better than the compared technique Ochiai in terms of the Exam score measure.

VI. Conclusion

The need of more advanced techniques for software fault localization requires more time and resources as software systems are turn out to be more convoluted and larger in scale. It is necessary that software engineers and system analysts involved in software development possess a good understanding of currently available techniques of fault identification and should have familiarity with the emerging trends in this important area. This paper attempts to review traditional and some recent works on software fault localization which will help researchers and software developers to better understand the developments in the area of software fault localization.

In this paper, we have emphasized on spectrum-based or lightweight fault localization because of its popularity in the field of software fault localization. We have explained with the help of a simple working example the process of spectrum-based fault localization by using the classic SBFL techniques like Tarantula and Ochiai.

Considering the ongoing developments in the field of SBFL we have illustrated two techniques which further improve the effectiveness of traditional SBFL methods in single fault context. The first technique makes use of failed execution slices to improve the fault detection efficiency of existing classic SBFL methods. The second technique works on the concept of suspiciousness of program entity and the suspiciousness of its fault context. The fault context of a program entity can be defined as the set of other program entities that were executed in the same failed execution apart from that program entity itself. This technique combines the suspiciousness of a program entity and the suspiciousness of its fault context to generate the final suspiciousness score which results in improved absolute rank of a faulty program entity.

This paper also includes a brief review on some standard metrics that are used to measure the effectiveness of fault localization techniques. The metrics mainly measure the effectiveness of fault localization technique in terms of how much code the developer needs to examine before locating the first faulty program entity such as a statement. The metrics discussed mainly include T-Score, the EXAM/ Expense score, P-Score, Top-N and Wilcoxon signed rank test.

The two approaches which are based on the concepts of Failed Execution Slice and Fault Contexts are evaluated experimentally on standard benchmarks Siemens programs to compare their effectiveness against the classic Ochiai method. Overall, the experiment results show that the two approaches further improve the performance of existing SBFL techniques significantly.

We believe that this paper will provide the software engineering community a wide ranging idea of the key issues pertaining to the demanding field of software fault localization and will also suggest new ideas for future research. For future work, our study will be extended to further improve the performance of existing SBFL techniques by investigating the impact of test suites on fault localization.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu & F. Wotawa, “A Survey on Software Fault Localization”, IEEE Transactions on Software Engineering, 42(8), 707–740, 2016.

[2] J. Krinke, Slicing, Chopping, and Path Conditions with Barriers. Software Quality Journal, 12(4), 339–360, 2004.

[3] J. S. Collofello & L. Cousins, “Towards automatic software fault localization through decision-to-decision path analysis,” in Proc. Nat. Comput. Conf., pp. 539–544, June 1987.

[4] M. Renieris & S. P. Reiss “Fault localization with nearest neighbour queries” in Proc. Int. Conf. Autom. Softw. Eng., Montreal, QC, Canada, pp. 30–39, 2003.

[5] J. A. Jones & M. J. Harrold “Empirical evaluation of the tarantula automatic fault-localization technique,” in Proc. Int. Conf. Autom. Softw. Eng., Long Beach, CA, USA, pp. 273–282, Nov. 2005.

[6] L. Naish, H. J. Lee & K. Ramamohanarao, “A model for spectra-based software diagnosis”, ACM Transactions on Software Engineering and Methodology, 20(3), 1–32, 2011.

[7] H. Ribeiro, P. Roberto de Araujo, M. Chaim, H. Souza and F. Kon, "Evaluating data-flow coverage in spectrum-based fault localization," in 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Porto de Galinhas, Recife, Brazil, pp. 1-11, 2019.

[8] W. E. Wong, & Y. Qi, “Effective program debugging based on execution slices and inter-block data dependency”, Journal of Systems and Software, 79(7), 891–903, (2006).

[9] X. Xu, V. Debroy, W. Eric Wong & D. Guo, “TIES WITHIN FAULT LOCALIZATION RANKINGS: EXPOSING AND ADDRESSING THE PROBLEM”, International Journal of Software Engineering and Knowledge Engineering, 21(06), 803–827, (2011).

[10] L. Zhao, L. Wang & X. Yin, “Context-Aware Fault Localization via Control Flow Analysis”, Journal of Software, 6(10), 2011.

[11] T. Shu, L. Wang and J. Xia, "Fault Localization Using a Failed Execution Slice," 2017 International Conference on Software Analysis, Testing and Evolution (SATE), Harbin, pp. 37-44, 2017.

[12] 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 ’11*, 2011.

[13] Yong WANG, Zhiqiu HUANG, Yong LI & Bingwu FANG, “Lightweight fault localization combined with fault context to improve fault absolute rank” *Sci. China Inf. Sci.*, vol. 60, no. 9, 2017.

[14] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, & S. P. Midkiff, “Statistical Debugging: A Hypothesis Testing-Based Approach”, IEEE Transactions on Software Engineering, 32(10), 831–848, 2006.

[15] W. E. Wong, V. Debroy, & D. Xu, “Towards Better Fault Localization: A Crosstab-Based Statistical Approach”, IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42(3), 378–396, 2012.

[16] Zunwen You, Zengchang Qin and Zheng Zheng, "Statistical fault localization using execution sequence" 2012 International Conference on Machine Learning and Cybernetics, Xian, pp. 899-905, 2012.

[17] A. Zeller & R. Hildebrandt, "Simplifying and isolating failure-inducing input", IEEE Transactions on Software Engineering, 28(2), 183–200, 2002.
[18] H. Cleve and A. Zeller, "Locating causes of program failures," Proceedings. 27th International Conference on Software Engineering, ICSE 2005., Saint Louis, MO, USA, 2005, pp. 342-351, 2005.
[19] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops" in Proc. Int. Conf. Autom. Softw. Eng., Long Beach, CA, USA, pp. 263–272, Nov. 2005.
[20] D. Jeffrey, N. Gupta and R. Gupta, "Fault localization using value replacement," in Proc. Int. Symp. Softw. Testing Anal., Seattle, WA, USA, pp. 167–178, Jul. 2008.
[21] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang and X. Wang, "Capturing propagation of infected program states," in Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng., New York, NY, USA, pp. 43–52, 2009.
[22] W. E. Wong & Y. Qi "BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION", International Journal of Software Engineering and Knowledge Engineering, 19(04), 573–597, 2009.
[23] P. Cellier, M. Ducasse, S. Ferre, & O. Ridoux "Formal concept analysis enhances fault localization in software," in Proc. Int. Conf. Formal Concept Anal., Montreal, QC, Canada, pp. 273–288, Feb. 2008.
[24] C. Mateis, M. Stumptner, and F. Wotawa, "Modeling Java programs for diagnosis," in Proc. Eur. Conf. Artif. Intell., Berlin, Germany, pp. 171–175, 2000.
[25] F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnose programs automatically" in Proc. Int. Conf. Ind. Eng., Appl. Artif. Intell. Expert Syst., Cairns, Qld., Australia, pp. 746–757, 2002.
[26] W. Mayer and M. Stumptner, "Approximate modeling for debugging of program loops" in Proc. Int. Workshop Principles Diagnosis, Carcassonne, France, pp. 87–92, 2004.
[27] A. Griesmayer, S. Staber, & R. Bloem, Fault localization using a model checker. Software Testing, Verification and Reliability, 20(2), 149–173, 2009.
[28] T. Ball, M. Naik, & S. K. Rajamani From symptom to cause. ACM SIGPLAN Notices, 38(1), 97–105, 2003.
[29] A. Groce, "Error explanation and fault localization with distance metrics" Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, USA, 2005.
[30] Liu & J. Han, "Failure proximity", Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering - SIGSOFT '06/FSE-14, 2006.
[31] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using DStar (D*)" in Proceedings of 6th International Conference of Software Security Rel., Washington, D.C., USA, pp. 21–30, 2012.
[32] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localization," in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2017. ACM, 2017, pp. 273–283.
[33] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," J. Syst. Softw., vol. 82, no. 11, pp. 1780–1792, 2009.
[34] Cherry Oo, and Hnin Min Oo, "Automatic Program Repair of Java Single Bugs using Two-level Mutation Operators," IAENG International Journal of Computer Science, vol. 47, no.2, pp223-233, 2020.
[35] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," Empir. Softw. Eng., vol. 10, no. 4, pp. 405–435, 2005.
[36] A. Zakari, S. P. Lee, K. A. Alam, and R. Ahmad, "Software fault localisation: a systematic mapping study," IET Softw., vol. 13, no. 1, pp. 60–74, 2019.
[37] A. Zakari, S. P. Lee, and I. A. T. Hashem, "A single fault localization technique based on failed test input," Array, vol. 3–4, no. 100008, p. 100008, 2019.