EVALUATING THE EFFECTIVENESS OF BEN IN LOCALIZING DIFFERENT TYPES OF SOFTWARE FAULT

by

JAGANMOHAN CHANDRASEKARAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2015

Copyright © by Jaganmohan Chandrasekaran 2015

All Rights Reserved

ACKNOWLEDGEMENTS

First I would like to extend my heartfelt thanks and gratitude to my thesis advisor Professor Dr.Jeff Lei for providing me an opportunity to work in his group, for his continuous guidance and helping me to evolve as a better researcher. Without his kindness, patience and support this work would not be possible. I would also like to thank Professor Dr. Hao Che and Professor Dr. Junzhou Huang for agreeing to be part of my thesis committee.

I would like to thank Mrs. Laleh Sh.Ghandehari for her guidance throughout my thesis work.

As always I am thankful to my Amma and Appa for their sacrifice, continuous support and motivation. I am deeply indebted to my friends – Vinoth, Raja, R.R.Dhinesh, Amarnath, Someshwar for their support during the course of my thesis work. I would like to thank my friends here at Arlington, Danny and Abhishek for their help, support and motivation.

Last but not least, I cannot thank Ankita enough for being my pillar of strength, by allowing me to chase my dreams and being supportive as always.

July 31, 2015

ABSTRACT

EVALUATING THE EFFECTIVENESS OF BEN IN LOCALIZING DIFFERENT TYPES OF SOFTWARE FAULT

Jaganmohan Chandrasekaran, MS

The University of Texas at Arlington, 2015

Supervising Professor: Jeff Lei

Debugging refers to the activity of locating software faults in a program and is considered to be one of the most challenging tasks during software development. Automated fault localization tools have been developed to reduce the amount of effort and time software developers have to spend on debugging. In this thesis, we evaluate the effectiveness of a fault localization tool called BEN in locating different types of software faults. Assuming that combinatorial testing has been performed on the subject program, BEN leverages the result obtained from combinatorial testing to perform fault localization.

Our evaluation focuses on how the following three properties of software fault affect the effectiveness of BEN: (1) Accessibility: Accessibility refers to the degree of difficulty to reach (and execute) a fault during a program execution; (2) Input-value sensitivity: A fault is input-value sensitive if the execution of the fault triggers a failure only for some input values but not for other input values; and (3) Control-flow sensitivity: A fault is control-flow sensitive if the execution of the fault triggers a failure while inducing a change of control flow in the program execution.

We conducted our experiments on seven programs from the Siemens suite and two real-life programs, *grep* and *gzip*, from the SIR repository. Our results indicate that BEN is very effective in locating faults of low accessibility. A unique aspect of BEN is that

it generates a failed test called the core member and a small group of passed tests called the derived members. The spectrum of the core member is compared to that of each derived member. The way in which the core and derived members are generated allows faults of low accessibility to be effectively located by BEN. Our results also suggest that in most of the cases, BEN is effective in locating input value- and control flow-insensitive faults. However, no conclusion can be drawn from the experimental data about the individual impact of input value sensitivity and control flow sensitivity on BEN's effectiveness.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Software debugging accounts for a major part of the expenditure in the software development cycle. Debugging refers to the activity of locating software faults in a program and is considered one of the most challenging tasks. A significant amount of effort is spent in locating faults during debugging [4]. Automated fault localization tools were developed to reduce the amount of effort and time the developers spend on debugging.

Spectrum-based fault localization is a commonly used fault localization technique. This technique is based on the notion of a program spectrum. A program spectrum records information about certain aspects of a test execution [5] such as program paths and function call counts [7]. In this fault localization technique, we identify faults by comparing the spectrums of failed and passed test executions. The program statements that are only exercised by failed tests are likely to be faulty [1].

Ghandehari et al. [2] reported a spectrum-based fault localization tool called BEN, which was developed at the University of Texas at Arlington. BEN leverages the results obtained from combinatorial testing to perform fault localization. It is assumed that a combinatorial test set has already been executed, and the result of each test execution is available before the application of BEN. The fault localization process in BEN involves two major phases. In the first phase, BEN identifies the failure-inducing combination from the initial combinatorial test set. A combination is considered to be failure-inducing, if it causes any test in which it appears to fail [2]. In the second phase, it generates a small group of tests containing one failed test called core member and a set of passed tests called derived members using the failure-inducing combination from the first phase. The core member and derived members are executed, and the spectrum of the core member

is compared with the spectrum of each of the derived members. Based on this comparison, statements are ranked in terms of their likelihood to be faulty [3].

The main goal of this thesis is to evaluate the effectiveness of BEN in locating different types of software fault. The effectiveness of BEN is measured based on the number of statements the user has to inspect to locate the faulty statement. If the user is required to inspect a less number of statements to locate the fault, then BEN is considered more effective. The evaluation focuses on how the following three properties of software fault affect the effectiveness of BEN: (1) Accessibility: Accessibility refers to the degree of difficulty to reach (and execute) a fault during a program execution; (2) Input-value sensitivity: A fault is input-value sensitive if the execution of the fault triggers a failure only for some input values but not for other values. This implies that the faulty statement could be executed by both failed and passed test cases; and (3) Control-flow sensitivity: A fault is control-flow sensitive if the execution of the fault triggers a failure while inducing a change of control flow in the program execution.

The experiments were performed on seven programs from the Siemens suite, and two real-life programs, grep and gzip, from the SIR [16]. Each of these programs has several faulty versions, and BEN is applied on all of these faulty versions. Ghandehari et al. [10] provided the input parameter modeling information and fault localization results of the subject programs, which were used in the experiments. A random test set-based approach was followed to determine the three properties of the fault in the subject programs. The first step in this approach was to generate a random test set based on the input parameter model of each subject program. In the next step, the random test set is executed on each subject program, with test executions being recorded. Then, the execution trace information was analyzed to determine the three properties of the fault as follows. Accessibility was measured as the ratio of the number of test cases that

executed the faulty statement to the total number of test cases in the random test set. A fault is considered to be input value sensitive if there exists at least one test case from the random test set that executes the faulty statement and produces a passed execution. To determine control flow sensitivity, we select the first failed test case from the test set results, and its faulty version execution trace was compared with its error-free version execution trace. If the comparison does not indicate the change in program flow, the next failed test case from the test set was selected and the procedure is repeated. If there exists at least one failed test case such that the fault induces a change in program flow and produces a failure, we consider the fault to be control flow sensitive and otherwise control flow insensitive

The experimental results suggest that BEN is highly effective in locating low accessibility faults. This is because low accessibility faults are likely to be executed only by the core member but not by the derived members. This will allow BEN's ranking algorithm to rank the corresponding faulty statements higher. On the other hand, if a fault is easy to access, the fault is likely to be exercised by both core and derived members. As a result, the corresponding faulty statement is ranked lower by BEN's ranking algorithm. The experimental results also suggest that in most of the cases, BEN is effective in locating input value insensitive and control flow-insensitive faults. However, with the experimental results, we were unable to reach a conclusion about the individual impact of input value sensitivity and control flow sensitivity on BEN's effectiveness in locating the fault.

The remainder of the thesis is organized as follows. In Chapter 2, a discussion is presented on BEN's approach to locating faults. We also explain how the effectiveness of BEN is measured. Chapter 3 discusses the three fault properties and approaches to determining these properties. Also in this chapter, detailed examples are provided for

each of the three properties of the fault. Chapter 4 presents information about the subject programs, including the input parameter model and fault localization results of each subject program. We also present a discussion on the approach we used to determine the fault properties in the subject programs. Chapter 5 presents the results of our experiments. In this chapter, we provide the classification of faults in the subject programs that were made using the approach discussed in Chapter 4. We also discuss the individual impact of the three properties in BEN's effectiveness in locating a fault. Chapter 6 discusses the related work about evaluating the effectiveness of fault localization techniques. In Chapter 7 we provide the concluding remarks and discuss several directions for our future work.

CHAPTER 2

BEN: A FAULT LOCALIZATION TOOL

2.1 Overview

BEN is a spectrum-based fault localization tool, which uses the result of a combinatorial test set. In spectrum based fault localization technique, the program spectrum, e.g. statement coverage, is recorded for each test and then the spectra of the failed and passed tests are compared to locate the fault. Statements which are executed exclusively by the failed test cases i.e. statements those appear only in the program execution spectra of the failed test cases and not appearing in the execution spectra of the pass test cases are considered to be highly suspiciousness and most-likely to be faulty. Fault localization methods like Tarantula [6], Set Union, Set Intersection, and nearest neighbor [7] also follows the spectrum-based approach. BEN is different from the other spectrum-based approaches in the way that the other spectrum-based methods assume an existence of a larger number of test runs and locates the fault by comparing the pass/fail spectra with respect to these test runs; where as, BEN uses the result of a combinatorial test set and generates a small number of tests to locate the fault. BEN's approach is inspired by the concept of the nearest neighbor method [1].

The fault localization process in BEN involves two major phases. In the first phase, it identifies failure-inducing combinations; failure-inducing combination is a combination, which if it appears in a test, cause the test to fail. In the second phase, BEN produces a ranking of statements in terms of their likelihood of being faulty. The user should investigate the statements in the top rank at first, and then go to the next rank (if the fault is not found in the top rank) until the fault is located.

2.2 Phase 1

At the start of phase 1, the user provides the following inputs (1). Subject program (2).A combinatorial test set created based on the input parameter model. It is assumed the user has executed the combinatorial test set, and the results (test execution status) of each test are known [3]. First, BEN analyzes the initial combinatorial test set and identifies a set of suspicious combinations and then ranks them based on their suspiciousness. A suspicious combination is a combination which is candidate of being inducing combination. For a given test set F, any combination that only appears in the failed test of F is considered suspicious. BEN ranks the suspicious combinations based on their likelihood to be inducing. Ranking of the suspicious combinations is based on the suspiciousness of combinations (SoC) and suspiciousness of the environment of the combinations (SoE). Suspiciousness of combination is computed based on suspiciousness of components (parameter values) appear in the combination. Suspiciousness of the environment for the combination is computed based on the suspiciousness of components that do not appear in the combination but appear in the same test as the combination. The suspicious combination, which has a higher SoC and lower SoE value in the set F, will be ranked higher.

Followed by the rank generation, BEN generates a small set of new test cases for top-ranked suspicious combinations, which the user can choose to execute and refine the suspicious combinations set. A new test is generated in a way that it contains the suspicious combination and has the minimum suspiciousness in the environment. If such a test, fails, it is very likely that the suspicious combination is inducing. Otherwise the suspicious combination is not suspicious any more since it appears in a passed test. The rank and test generation continues iteratively until the stopping condition is satisfied [2]. At the end of Phase 1, BEN identifies the failure-inducing suspicious combination.

2.3 Phase 2

BEN generates a small group of tests S based on the failure-inducing combination identified at Phase 1. The set S will contain one core member, that is a failed test and several derived members that are passed tests. BEN generates the core member such that it contains the failure-inducing combination. The derived members are very similar to the core member, however, they do not contain the failure-inducing combination, and they are expected to produce a program execution trace that is similar to core member execution trace. Once, the test generation is completed, the user is expected to execute them. The core member is very likely to fail as it contains the failure-inducing combination. Likewise, all the derived members on test execution are most likely to result in a pass, as they do not contain the failure-inducing combination. If the core member does not result in a failure, BEN will select a failed test case from the initial test set as a core member.

During the test execution, the program spectra for the core and derived members are recorded with the help of coverage tool GCOV. BEN compares the execution trace of the core member (failed test case) with each derived member's execution trace to compute the suspiciousness score of every statement. If a derived member does fail on test execution, BEN will ignore that derived member and will not consider it, for comparison with the core member.

The suspiciousness score represents how likely the statements is faulty. Statements executed only by the core member are considered most suspicious, and BEN will rank them higher. Statements executed by both core and derived member are considered less suspicious. The statements executed only the derived member is considered non-suspicious. Statements are ranked in descending order of their suspiciousness value.

User will locate the fault, by investigating the statements in the top Rank at first and user might continue to investigate (if fault not located in top ranked statements) the statement in the next rank, until the faulty statement is found. The effectiveness of BEN is measured based on the number of statements, a user has to investigate to reach the faulty statement i.e. the ratio of number of statements to be investigated to locate the fault to the total number of statements in the subject program.

CHAPTER 3

FAULT PROPERTIES

This chapter presents the definition of three fault properties 1). Accessibility 2). Input value sensitivity and 3). Control flow sensitivity

3.1 Accessibility

Accessibility refers to the degree of difficulty to reach (and execute) a fault during a program execution. It is a value between 0 and 1. Accessibility is measured as the ratio of the number of test set executes the faulty statement to the total number of tests in the exhaustive test set.

If a faulty statement has an accessibility value of 1, it indicates that all the test cases do execute the faulty statement. The lesser the accessibility value, it is harder to execute the faulty statement. For example, a value of 0.2 indicates that only 20% of the total number of test cases executes the faulty statement. In testing phase, low accessibility faults are difficult to detect, as the small number of tests could reach and execute the faulty statement.

Example:

Consider the example in Table 3-1: In this program, if the customer makes a purchase > $1000, a discount will be applied based on the type of membership. EXECUTIVE members will receive 25% discount, GOLD members will receive 10%, and customers who do not have a membership will receive 5% discount. There are three inputs for the method *applyDiscount()* and they are amount of purchase made by the customer, membership information of the customer i.e. is the customer holds an membership(Y or N); If yes, the type of membership i.e. either executive or gold membership('E' or 'G').

Table 3-1 - Example for Accessibility

```
1.  public int applyDiscount(int totalPrice,bool member, char
    type)
2.  {
3.    float discount = 0.00;
4.   if(totalPrice>100) //Fault #1 - correct :if(totalPrice>1000)
5.   {
6.     if(member == TRUE)
7.      {
8.        if(type == “E”)
9.         {
10.          discount = (0.25)*totalPrice;
11.          totalPrice = totalPrice-Discount;
12.        }
13.        if(type == “G”)
14.        {
15.          //Fault # 2 - correct : Discount = (0.10)*totalPrice;
16.          Discount = (0.07)*totalPrice;
17.          totalPrice = totalPrice-Discount;
18.        }
19.     }
20.     else
21.      {
22.       discount = (0.05)*totalPrice;
23.       totalPrice = totalPrice-Discount;
24.     }
25.   }
26. }
```

This program has two faults. Fault (F1) at line # 4 and Fault (F2) at line # 16

- Fault - F1: - A discount should be applied to customers whose purchase value is >$1000; instead discount will be applied for purchases > $100.
- Fault -F2: - For Gold Members, 7% of discount applied instead of 10% discount.

We use an exhaustive test set to perform testing. The testing and results are summarized in Table 3 - 2.

Table 3-2 - Test Case and Results for Accessibility Example

Test Case				Result
Test case #	totalPrice	member	type	
T1	500	T	E	Fail
T2	500	T	G	Fail
T3	500	F	null	Fail
T4	10	T	E	Pass
T5	10	T	G	Pass
T6	10	F	null	Pass
T7	1500	T	E	Pass
T8	1500	T	G	Fail
T9	1500	F	null	Pass

The Table 3-2 summarizes the exhaustive test set and results. For our example, there are nine test cases. The execution trace of this exhaustive test set is provided in Appendix B.

Fault 1 at line # 4 (refer Appendix B) is executed by all the test cases. This fault is located at the start of the program; so all the nine test cases execute the fault. So, the accessibility value for this faulty statement is (9/9) = 1. This value indicates the faulty statement has a higher degree of accessibility. i.e. it is accessed by all the test cases from the exhaustive test set.

For a test case to access the faulty statement (F2) at line # 16 (refer Appendix B), the test case has to satisfy the following criteria

1. Total Price > 1000 AND
2. Membership = True AND
3. Membership type = “G”

Among the nine test cases from the exhaustive test set, only T8 satisifies the critieria. However, because of fault #1, Test case T2 is able to access this fault. Therefore, the accessibility value for this faulty statement is calculated (2/9) = 0.22 (refer Appendix B). This value indicates a lower degree of accessibility i.e. not many test cases from the exhaustive test set will have access to this fault.

3.2 Input Value Sensitivity

A fault is considered to be input-value sensitive if the execution of the fault triggers a failure only for some input values but not for other values. i.e., executing a fault will not cause failure for all input values. This kind of fault, on execution, can result in a pass state for certain input values.

Example: -

Table 3-3 - Program to Calculate Sum of Two Numbers

```
1.  public int sum(int num1, int num2)
2.  {
3.      int sum=0;
4.     //fault:  operator  " * " is used instead of "+"
5.      sum = num1 * num2;
6.      return sum;
7.  }
```

Consider the program, which calculates the sum of two numbers and stores the result in a variable *sum.* This program has a fault at line # 5. The program will perform the multiplication of two numbers instead of addition. The testing and the results are summarized in the following table (refer Table 3-4).

Table 3-4 - Test Case and Results for Input Value Sensitivity Example

Test Case			Result
Test case #	Input Value 1	Input Value 2	
T1	2	2	Pass
T2	2	3	Fail
T3	1	1	Fail

Table 3-5 - Execution Trace Information for Input Value Sensitivity Example

		T1	T2	T3
1.	public int sum(int num1, int num2)	•	•	•
2.	{	•	•	•
3.	int sum=0;	•	•	•
4.	//fault: operator fault - "*" is used instead of "+"			
5.	sum = num1 * num2;	•	•	•
6.	return sum;	•	•	•
7.	}	•	•	•

Table 3-5 represents the execution trace of three test cases - T1, T2, and T3. The executed statements are marked as "•". The table indicates all the test cases (T1, T2, and T3) execute the faulty statement (at line # 5). The faulty statement, on execution triggers a failure for test cases T2 and T3. Test case - T1 executes the faulty statement,

yet does not produce a failure. During the execution of T1, the fault is executed, and the program transition to an infectious state but the infection does not propagate to the failure. This is known as “Co-incidental Correctness”.

As, this fault triggers a failure only for certain input values and not for all input values. We consider this fault as *input value sensitive fault*.

3.3 Control Flow Sensitivity

A fault is considered to be control-flow sensitive if the execution of the fault triggers a failure while inducing a change of control flow in the program execution.

Example: -

Table 3-6 - Example for Control Flow Sensitivity fault

```
1.    void printRange(int number)
2.    {
3.    // fault: “ > “ is used instead of “ >= “
4.    if(number>10)
5.    {
6.       printf(“The number is > 9”);
7.    }
8.    else
9.    {
10.      printf(“The number is < = 9”);
11.   }
12.   }
```

Consider the example in Table 3-6. This program will display a message “*The number is > 9*” if the input is greater than 9. If the input is between 0 to 9, the program will display a

message “*The number is* <=*9*”. The below table (refer Table 3-7) summarizes the test cases and results.

Table 3-7 -Test Case and Results for Control Flow Sensitivity Example

Test Cases		Result
Test #	Input Value	
T1	12	Pass
T2	5	Pass
T3	10	Fail

The fault is at line # 4. When test case “T3” with an input value of 10, executes the faulty statement, the fault induces a change in program flow and triggers a failure. The system incorrectly displays “*The number is* < =*9*”. The following table (Table 3-8) summarizes the expected and actual program flow for test case “T3”.

Table 3-8 - Execution Trace (Expected Vs Actual) For Test Case - T3

Line #	Statements	Testcase - T3 (input value = 10)	
		Expected Program Flow	Actual Program flow
1.	void printRange(int number)	•	•
2.	{	•	•
3.	// fault: " > " is used instead of " >= "		
4.	if(number>10)	•	•
5.	{	•	
6.	printf("The number is > 9");	•	
7.	}	•	
8.	else		•
9.	{		•
10.	printf("The number is < = 9");		•
11.	}		•
12.	}	•	•

Faulty statement (line # 4) on execution, induce a change in program flow for test case - T3. The program changes its flow; skips line # 5 - line #7 and it executes line # 8 - line# 11. This fault induces a change in program flow and results in a failure. We consider this type of fault as a control-flow sensitive fault.

CHAPTER 4

EXPERIMENTAL DESIGN

This chapter discusses subject programs and their input parameter modeling, fault localization results of subject programs, methodology and metrics.

4.1 Subject Programs

For our experimental study, we have used the Siemens suite and two real-life large programs grep and gzip [16]. In this section, we discuss *Siemens* suite, grep and gzip in detail.

4.1.1 Siemens suite

The Siemens suite consists of 7 benchmark programs namely *schedule, schedule2, printtokens, printtokens2, totinfo, tcas and replace*. Each of the programs in the *Siemens* suite contains several faulty versions. The table below (Table 4-1) shows the number of faulty versions and the number of lines of the executable code for each program. The number of executable code is computed using GCOV.

Table 4-1 - Siemens Suite

Programs	Number of Faulty Versions	Lines of Executable Code
Schedule	9	154
Schedule2	10	127
printtokens	7	188
printtokens2	10	201
totinfo	23	123
tcas	41	65
replace	32	242

4.1.2 Grep and Gzip

Siemens suite consists of programs that are smaller in size (Lines of executable code). Therefore, in addition to the Siemens suite, we have selected two real-life programs (larger programs) namely GREP and GZIP from the SIR repository for our experiments. Both grep and gzip have five different implementations, and each implementation has several faulty versions. The table below summarizes the *GREP* and *GZIP* programs, the number of faulty versions of all five different implementations.

Table 4-2 - Grep and Gzip

Programs		Number of Faulty Versions	Lines of Executable code
GREP	grep1	18	3078
	grep2	8	3224
	grep3	18	3294
	grep4	12	3313
	grep5	1	3314
GZIP	gzip1	16	1705
	gzip2	7	2006
	gzip3	10	1866
	gzip4	12	1892
	gzip5	14	1993

4.2 Input Parameter Modeling for Subject Programs

The input model information for each subject program is summarized in Table 4-3.The *Model* column in Table 4-3 represents the number of parameters and their domain size. Each model is represented by (d_1^{p1} x d_2^{p2} x d_3^{p3} x d_4^{p4} x …) where d_1^{p1} indicates there are p1 number of parameters with a domain size of d1. For example, the program

replace has 20 parameters in total. There are four parameters with a domain size of 2 and 16 parameters with a domain size of 4. The detailed information about Input parameter modeling for the subject program is available in [8]. For all subject programs, the combinatorial test sets with strength 2 to 6 were created using the ACTS tool [9]. All faulty versions of each subject program were tested using 2-way test set. If the faulty version is not killed by the test set, i.e. no failed test is found, we increased the strength of the combinatorial test set until it is killed, or the strength of combinatorial test set becomes 6. Table 4-4 indicates the number of killed versions of each program. We ignored faulty versions that are not killed. If no failed test is found, no need to use fault localization tool.

Table 4-3 - Input Parameter Model for Subject Programs

Programs	Model	Number of Constraints
printtokens	$(2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2)$	8
printtokens2	$(2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2)$	8
replace	$(2^4 \times 4^{16})$	36
schedule	$(2^1 \times 3^8 \times 8^2)$	0
schedule2	$(2^1 \times 3^8 \times 8^2)$	0
tcas	$(2^7 \times 3^2 \times 4^1 \times 10^2)$	0
totinfo	$(3^3 \times 5^2 \times 6^1)$	0
grep	$(2^7 \times 4^1 \times 5^1 \times 6^3 \times 8^1 \times 9^1 \times 13^1)$	1
gzip	$(2^{11} \times 4^2)$	8

Table 4-4 - Test results for subject programs

Programs	Total # of Faulty Versions	# of faulty versions used in our experiments
Schedule	9	7
Schedule2	10	3
printtokens	7	3
printtokens2	10	9
totinfo	23	12
tcas	41	36
replace	32	32
grep1	18	4
grep2	8	4
grep3	18	7
grep4	12	2
grep5	1	0

Table 4-4 - continued

gzip1	16	7
gzip2	7	3
gzip3	10	0
gzip4	12	3
gzip5	14	4

4.3 Fault Localization Results

The effectiveness of BEN is measured based on the percentage of the source code the user has to inspect to reach the fault. Step one in our experiment is to perform fault localization on the subject programs using BEN and determine BEN's effectiveness for all faulty versions of all subject programs. This information forms the basis for our experiment; Ghandehari et al. [10] has performed a similar experiment in which BEN was used to locate the fault in the subject programs; they have shared the results, which has been used in our experiments.

4.4 Methodology

For our experiments, we need to determine the three properties of the faults in our subject programs. Section 3.1 suggests that an exhaustive test set is required to measure the accessibility. However, for our subject programs, it is not possible to create exhaustive test set. Therefore, we use a random test set in place of an exhaustive test set. In this approach, we generate the random test set for each subject program based

on their input parameter model, execute them on our subject programs and evaluate the results to determine three fault properties in our subject program.

Note that if two programs share the same input parameter model, they also share the same random test set. Thus, we have generated nine random test sets (as we have nine different input parameter model Table 4-3), each having maximum 1000 test cases. If the input parameter model does not have any constraint, a random test is created in a way that for each parameter, we randomly select one of the possible values from its corresponding domain. If the model has a constraint, a random test is created in a way that it will not violate the constraint. More information about random test generation can be found at [18].

There are certain subject programs, which had an exception. GZIP has complex constraints; it prevented us from generating 1000 unique random test cases. The maximum unique random test cases we were able to generate were 395. For tcas program from the Siemens suite, there are eight faulty versions for which, the random test set on execution does not generate a single fail test. Therefore, we have used an additional test case (failed) from their respective initial test set. For these eight versions, we have used 1001 test cases. The next step is to execute the random test set on each faulty version (see Table 4-3) of the subject program; we used GCOV to record the test case execution.

4.4.1 Accessibility

Once the random test set execution on each faulty version is completed, we will use the GCOV file to measure the accessibility. For each subject program, the accessibility value for all the faulty versions calculated using the approach discussed in Chapter 3.1 will give us a continuous data, and there is a possibility of an infinite number of possible values within the range 0 to 1. So, we have used MEDIAN score to divide the

faulty versions into two groups i.e. faulty with high accessibility or faulty version with low accessibility. Any faulty version with an accessibility value < MEDIAN, will be considered to have low accessibility (harder to access) and Faulty version with an accessibility value > = MEDIAN, is considered to have high accessibility (easier to access). We followed this approach for all subject program used in our experiments.

4.4.2 Input Value Sensitivity

Identification of input value sensitivity property of a fault is a two-step process. In step 1, we will use the GCOV file to identify the test cases that execute the faulty statement. In the next step, we identify the input value sensitivity using the approach discussed in Chapter 3.2 i.e. If there exist, at least one test case that on execution of the faulty statement, produced a pass state, we will consider the fault as input value sensitive fault. If all the test cases that execute the faulty statement produce a fail state, we will consider the fault as input value insensitive fault.

4.4.3 Control Flow Sensitivity

To determine the control flow sensitivity property of the fault, in step 1 we will execute the random test set on the error-free version of the subject program and record the test case execution. In step 2, a failed test case was randomly selected from the test set results, and its faulty version execution trace was compared with its error-free version execution trace. If the comparison does not indicate the change in program flow, another failed test case was randomly selected from the test set results and the procedure is repeated. If there exist at least one failed test case, which on comparison, suggests that the fault induces a change in program flow and produces a failure, we consider the fault to be control flow sensitive. If all the failed test cases in the random test set, on comparison suggest, the fault does not induce a change in program flow and yet trigger a

failure, we consider them as control flow-insensitive faults. Appendix A summarizes the classification of faults based on three properties discussed above.

4.5 Metrics

The goal of our experimental design is to answer the following research questions by empirical studies.

- R1: How does Accessibility affect the BEN's effectiveness in locating the fault?
- R2: How does input value sensitive software fault affect the BEN's effectiveness in locating the fault?
- R3: How does control flow-sensitive software fault affect the BEN's effectiveness in locating the fault?

CHAPTER 5

EXPERIMENTAL RESULTS

Table 5-1 shows the overall results of the BEN's effectiveness in locating different fault types.

Table 5-1 - Overall Results

Input Value Sensitivity	Control Flow Sensitivity	Accessibility	# Of faulty versions	Average % of code to be diagnosed
Y	Y	H	78	19.73
Y	Y	L	21	4.29
Y	N	H	2	19.59
Y	N	L	3	18.89
N	Y	H	4	8.06
N	Y	L	15	4.52
N	N	H	0	NA
N	N	L	3	5.96

Totally, 126 fault versions were used in our experiments. In the table above, column "*# Of Faulty Versions"* indicates the number of faulty versions belongs to the particular category, Column 1 & 2 represents the fault properties Input Value Sensitivity and Control flow sensitivity respectively. "Y" indicates that the faulty versions exhibit the respective property and "N" indicates the faulty versions do not exhibit that respective property. Column 3 represents the Accessibility property. A value of "H" denotes high accessibility to the fault and "L" denotes low accessibility to the fault. Following is the brief summary of faulty versions from the table 5 -1

- 78 faulty versions that contain input value sensitive, control flow-sensitive and easier to access software fault.
- There is not a single faulty version that contains an input value and control flow-insensitive fault with high accessibility.

In this thesis, our main goal is to evaluate how three properties affect the BEN's effectiveness in locating a fault. In the following three sections, we discuss how each property (accessibility, input value sensitivity and control flow sensitivity) affect the BEN's effectiveness in locating the fault

5.1 Impact of Accessibility

To determine the impact of accessibility on BEN's effectiveness in locating the fault, we fix the other two properties and analyze the results i.e. we will select and compare the effectiveness of faulty versions, whose Input Value Sensitive and Control Flow sensitive values are fixed. We will have four sets of combinations as listed below

1. Input Value sensitive and Control Flow sensitive fault
2. Input value sensitive and Control flow insensitive fault
3. Input value insensitive and Control flow sensitive
4. Input value and Control flow insensitive fault

Since the other two factors are fixed, we believe any variation in BEN's performance is due to the change in Accessibility. This is the main idea behind our approach.

5.1.1 - Set 1 - Input Value Sensitive and Control Flow Sensitive

The following table summarizes set 1.

Table 5-2 - Accessibility Set 1

Input Value Sensitivity	Control Flow Sensitivity	**Accessibility**	# of versions	Average % of code to be diagnosed
Y	Y	**H**	78	19.73
Y	Y	**L**	21	4.29

For set 1, we consider the test versions that have a software fault, which is input value sensitive as well as control flow sensitive. In set 1, on an average, to locate a input value and control flow sensitive fault which is easier to access (high accessibility), the user is expected to diagnose 19.73% of the source code whereas, to locate input value sensitive and control flow fault which is harder to access (low accessibility), the user is expected to diagnose 4.29% of the source code on an average. On comparison, we conclude, BEN has a better effectiveness in locating an input value sensitive and control flow software fault that are harder to access (low accessibility).

5.1.2 - Set 2 - Input Value Sensitive and Control Flow Insensitive

Table 5-3 - Accessibility Set 2

Input Value Sensitivity	Control Flow Sensitivity	**Accessibility**	# of versions	Average % of code to be diagnosed
Y	N	**H**	2	19.59
Y	N	**L**	3	18.89

For set 2, we consider the test versions that consist of software faults, which are input value sensitive and control flow insensitive. For this set, on an average, to locate a input value sensitive and control flow insensitive fault with high accessibility, the user is expected to diagnose 19.59% of the source code whereas, to locate input value sensitive and control flow insensitive fault which is harder to access (low accessibility), the user is expected to diagnose 18.89% of the source code on an average. By comparing the two, though the difference is very minimal, we conclude that BEN is effective in locating input value sensitive and control flow insensitive fault that are harder to access (low accessibility).

5.1.3 - Set 3 - Input Value Insensitive and Control Flow Sensitive

Table 5-4 - Accessibility Set 3

Input Value Sensitivity	Control Flow Sensitivity	**Accessibility**	# of versions	Average % of code to be diagnosed
N	Y	**H**	4	8.06
N	Y	**L**	15	4.52

For Set 3, we consider the test versions, which contains input value insensitive and control flow sensitive software fault. For this set, on an average, to locate an input value insensitive and control flow sensitive fault with a high accessibility, the user is expected to diagnose 8.06% of the source code to locate the fault; whereas to locate an input value insensitive and control flow sensitive fault with low accessibility, the user is expected to diagnose 4.52% of the source code. The result clearly suggests that BEN is effective in locating input value insensitive and control flow sensitive fault with low accessibility.

5.1.4 - Set 4 - Input Value and Control Flow Insensitive

Table 5-5 - Accessibility Set 4

Input Value Sensitivity	Control Flow Sensitivity	**Accessibility**	# of versions	Average % of code to be diagnosed
N	N	**H**	0	NA
N	N	**L**	3	5.96

For set 4, we consider the test versions, which contains input value and control flow insensitive software fault. Based on our classification of benchmark programs, we do not have at least 1 version that belongs to the category non-input value and control flow sensitive with higher accessibility (N, N, H). So, we were unable determine on how higher accessibility will impact the BEN effectiveness in locating input value and control flow insensitive fault. The results suggest, to locate an input value and control flow insensitive fault with low accessibility, the user is expected to diagnose 5.96% of the source code.

Based on our observations from set 1, set 2, set 3, we believe, BEN is more effective in locating low accessibility faults than high accessibility faults. Theoretically, for faults that are harder to access, the probability of derived members executing them is very low. When the derived member does not execute the fault, 1). The chances of the faulty statement appearing only in the failed test case spectra are high; 2). BEN will have more passed test cases and 3). Based on BEN ranking algorithm, the faulty statement will be ranked higher. Experimental results also suggest the same. BEN effectively locates the faults that are harder to access.

BEN's approach is inspired by the notion of nearest neighbor [1]. A derived member spectrum is almost similar to the core member spectra. So, if the fault is easier to access, there is a high probability for derived member(s) executing the faulty statement. This results in BEN considering the faulty statement less suspicious and ranking them lower. Experimental results suggest the same. BEN has a low effectiveness in locating faults that are easier to access.

5.2 Impact of Input Value Sensitivity

To determine the impact of input value sensitivity on BEN's effectiveness in locating the fault, we fix the other two properties and analyze the results i.e. we will select and compare the effectiveness of faulty versions, whose Accessibility and Control Flow sensitive values are fixed. So, we will have four sets with the following combinations listed below

1. Control flow sensitive and High Accessibility
2. Control flow sensitive and Low Accessibility
3. Control flow-insensitive and High Accessibility
4. Control flow-insensitive and Low Accessibility.

Since the other two factors of the fault are fixed, we believe any variation in BEN's effectiveness is because of the Input Value Sensitivity property of the fault.

5.2.1 - Set 1 - Control Flow Sensitive and High Accessibility

Table 5-6 - Input Value Sensitivity - Set 1

Input Value Sensitivity	Control Flow Sensitivity	Accessibility	# of versions	Average % of code to be diagnosed
Y	Y	H	78	19.73
N	Y	H	4	8.06

In set 1, we consider the faulty versions, which contain a control flow sensitive fault with high accessibility. For this set, to locate a control flow sensitive, easier to access (High Accessibility) and *input value sensitive fault*, on an average the user is expected to diagnose 19.73% of the source code where as, to locate a fault which is control flow sensitive, easier to access (High Accessibility) and *input value insensitive*, the user is expected to diagnose 8.06% of the source code. The result suggests. BEN is highly effective in locating *input value insensitiv*e and control flow sensitive software fault with high accessibility.

5.2.2 - Set 2 - Control Flow Sensitive and Low Accessibility

Table 5-7 - Input Value Sensitivity - Set 2

Input Value Sensitivity	Control Flow Sensitivity	Accessibility	# of versions	Average % of code to be diagnosed
Y	Y	L	21	4.29
N	Y	L	15	4.52

In set 2, we consider the faulty versions, which contains a harder to access (low accessibility), control flow sensitive faults. For this set, to locate an *input value sensitive* and Control flow sensitive fault, which is harder to access (low accessibility); on average the user is expected to diagnose 4.29% of the source code. In comparison, to locate a *input value insensitive* and Control flow sensitive, that is harder to access (low Accessibility), the user is expected to diagnose 4.52% of the source code. The result suggests, BEN is effective in locating *input value sensitive* and control flow sensitive fault, that is harder to access (Low Accessibility).

5.2.3 - Set 3 - Control Flow Insensitive and High Accessibility

Table 5-8 - Input Value Sensitivity - Set 3

Input Value Sensitivity	Control Flow Sensitivity	Accessibility	# of versions	Average % of code to be diagnosed
Y	N	H	2	19.59
N	N	H	0	NA

In set 3, we consider the faulty versions, which contains a control flow-insensitive fault with high accessibility. Based on our classification of subject programs, we do not have at least 1 version that belongs to the category input value and control flow insensitive with higher accessibility (N, N, H). So, we were unable determine on how *input value sensitivity* will affect the BEN effectiveness in locating control flow insensitive fault with high accessibility. Our result suggests that to locate an input value sensitive and control flow insensitive fault with high accessibility (easy to access the fault), on an average the user is expected to diagnose 19.59% of the source code.

5.2.4 - Set 4 - Control Flow Insensitive and Low Accessibility

Table 5-9 - Input Value Sensitivity - Set 4

Input Value Sensitivity	Control Flow Sensitivity	Accessibility	# of versions	Average % of code to be diagnosed
Y	N	L	3	18.89
N	N	L	3	5.96

In set 4, we consider the faulty versions, which contains a control flow-insensitive fault with low accessibility. In this set, to locate an *input value sensitive* and Control flow-insensitive fault with low accessibility on an average the user is expected to diagnose 18.89% of the source code. In comparison, to locate an *input value insensitive* and Control flow insensitive fault with Low Accessibility, the user is expected to diagnose 5.96% of the source code. The result suggests BEN is effective in locating *input value insensitive* and Control flow insensitive with low accessibility.

To our surprise, result from one of the cases (set 2) suggests BEN is effective in locating an *input sensitive* fault. For the remaining two cases (set 1 and set 4), results suggest BEN is effective in locating *input value insensitive* fault; for set 3, we were unable to reach a conclusion about the impact of *input value sensitivity* on BEN's effectiveness in locating a control flow insensitive and high accessibility fault. In majority of the cases, the results suggest that BEN is effective in locating *input value insensitive* fault. However, there exist one set (set 2), in which BEN is effective in locating an input value sensitive fault. Overall, we were unable to draw conclusions about the impact of *input value sensitivity* on BEN's effectiveness in locating a software fault.

5.3 Impact of Control Flow Sensitivity

To determine the impact of control flow sensitivity on BEN's effectiveness in locating the fault, we fix the other two properties and analyze the results i.e. we will select and compare the effectiveness of faulty versions, whose Accessibility and Input value sensitive values are fixed. We will have four sets with the following combinations listed below

1. Input value sensitive and high accessibility
2. Input value sensitive and low accessibility
3. Input value insensitive and high accessibility
4. Input value insensitive and low accessibility

Since the other two factors of the fault are fixed, we believe any variation in BEN's effectiveness is because of the Control flow Sensitivity property of the fault.

5.3.1 - Set 1 - Input Value Sensitive and High Accessibility

Table 5-10 - Control Flow Sensitivity - Set 1

Input Value Sensitivity	**Control Flow Sensitivity**	Accessibility	# of versions	Average % of code to be diagnosed
Y	**Y**	H	78	19.73
Y	**N**	H	2	19.59

In set 1, we consider the faulty versions, which contains an input value sensitive, easier to access (high accessibility) faults. In this set, on an average, to locate a input value sensitive, easier to access and *control flow sensitive fault* the user is expected to diagnose 19.73% of the source code where as, to locate a input value sensitive, easier to access and *control flow insensitive fault*, the user is expected to diagnose 19.59% of the

source code on an average. On comparison, we conclude BEN has relatively better effectiveness in locating a input value sensitive, *control flow insensitive* and easier to access software fault.

5.3.2 - Set 2 - Input Value Sensitive and Low Accessibility

Table 5-11 - Control Flow Sensitivity - Set 2

Input Value Sensitivity	**Control Flow Sensitivity**	Accessibility	# of versions	Average % of code to be diagnosed
Y	**Y**	L	21	4.29
Y	**N**	L	3	18.89

In set 2, we consider the faulty versions, which contains an input value sensitive and harder to access (low accessibility) faults. In this set, on an average, to locate a input value sensitive, harder to access and control flow sensitive fault the user is expected to diagnose 4.29% of the source code where as, to locate input value sensitive, harder to access and control flow insensitive fault, the user is expected to diagnose 18.89% of the source code to locate the fault. On comparison, BEN has a relatively better effectiveness in locating a fault, which is input value sensitive, *control flow sensitive* and easier to access.

5.3.3 - Set 3 - Input Value Insensitive and High Accessibility

Table 5-12 - Control Flow Sensitivity - Set 3

Input Value Sensitivity	**Control Flow Sensitivity**	Accessibility	# of versions	Average % of code to be diagnosed
N	**Y**	H	4	8.06
N	**N**	H	0	NA

In set 3, we consider the faulty versions, which contains input value insensitive, high accessibility fault. In our subject programs, we do not have at least one version that belongs to the category input value and control flow insensitive with higher accessibility (N, N, H). So, we were unable determine on how *control flow sensitivity* property will affect the BEN effectiveness in locating input value sensitive fault with high accessibility. The results suggest, to locate an input value insensitive and control flow sensitive fault with high accessibility, on an average to locate the fault, the user is expected to diagnose 8.06% of the source code.

5.3.4 - Set 4 - Input Value Insensitive and Low Accessibility

Table 5-13 - Control Flow Sensitivity - Set 4

Input Value	**Control Flow**	Accessibility	# of versions	Average % of code to be diagnosed
N	**Y**	L	15	4.52
N	**N**	L	3	5.96

In set 4, we consider the faulty versions, which contains input value insensitive, harder to access (low accessibility) faults. In this set, on an average, to locate a input value insensitive, harder to access and *control flow sensitive fault* the user is expected to diagnose 4.52% of the source code where as, to locate input value insensitive, harder to access and *control flow insensitive fault*, the user is expected to diagnose 5.96% of the source code to locate the fault. On comparison, BEN relatively has a better effectiveness in locating a fault, which is input value insensitive, *control flow sensitive* and harder to access.

For certain scenarios(set 2 and set 4), the BEN has a better effectiveness in locating a *control flow-sensitive fault*. However, for set 1 our results suggest the opposite, i.e. BEN has a better effectiveness in locating *control flow-insensitive fault.* So, based on the overall results we were unable to draw a conclusion on the impact of control flow sensitivity on BEN's effectiveness in locating the fault.

CHAPTER 6

RELATED WORK

There are many existing empirical evaluations that have studied the effect of different factors on the effectiveness of fault localization techniques. Abreu et al. [11] discussed the effect of adding more failed and passed runs to the approach. Their result concludes that the effect of adding more passed runs is unpredictable, whereas the effectiveness improves if more failed test cases are used. Their result also suggests that fewer instances of coincidental correctness improve the overall effectiveness of the approach. Coincidental correctness is an event in which a test case executes the fault but coincidentally it does not result in a failure.

Jones et al. [6] compared the effectiveness of different fault localization techniques including Tarantula [19], Set intersection [26], Set Union [27], Nearest neighbor [7] and Cause transition [20]. In their experiment, the effectiveness is measured based on the percentage of the subject program that needs not be examined to find a faulty statement. Similar to our approach, they also follow the same methodology to distinguish between a failed and passed test case, i.e. First, they execute the test cases on the error-free version of the subject programs. These results would then be considered as the expected output. In the next step, they execute the test cases on the faulty versions, and the output is compared with their corresponding expected output. Their result suggests that set-intersection performs the worst while Tarantula performs the best in locating the faults. Their work is primarily focused on evaluating the effectiveness of different fault localization techniques, whereas our work is focused on evaluating the effectiveness of a particular fault localization technique in locating different fault types.

Baudry et al. [13] discussed the type of information needed for effective fault localization. Their work proposed a test-for-diagnosis criterion, which evaluates the capacity of the test cases to help fault localization. Their work discussed an explicit connection between testing and diagnosis. They introduced an attribute known as *Dynamic Basic Block (DBB)*, which is a set of statements that is covered by the same test cases in a test set. All the program statements in the same basic block will have the same rank. They propose that *DBB* is a decisive factor in the diagnosis accuracy of a fault localization technique. Their results suggest that increasing the number of *DBBs* improves the overall effectiveness of fault localization. They also confirm a strong correlation between the size of the *DBB* and the diagnosis accuracy. Faults that are located in small *DBBs* are effectively localized. Their work is different from ours, because we focus on how the fault properties that impact the effectiveness of fault localization technique, whereas their work is focused on improving the capacity of test cases in order to improve the effectiveness of fault localization.

Yu et al. [14] studied the impact of various test suite reduction strategies on the effectiveness of fault localization. They used a vector based reduction strategy and a statement-based reduction strategy for their study. In the vector based reduction strategy, the reduced test suite covers the same set of statement vectors as the original test suite. A statement vector is defined as the set of statements executed by one test case. The statement-based reduction strategy uses a reduced test suite that covers the same statements as the original test suite. Their experiment investigates the effect of test reduction strategies on the following fault localization techniques, including Tarantula [19], Statistical Bug Isolation [25], Jacaard [11] and Ochiai [17]. They have used the Siemens suite [24] as their subject programs. Their results show that the vector-based reduction has a negligible effect on the effectiveness, statement based reduction

significantly reduces the effectiveness. This work is focused on studying the effect of test case reduction in fault localization effectiveness, whereas our work is focused on understanding the effect of fault properties in fault localization effectiveness.

Renieris et al. [15] discussed the effects of two types of spectra, namely binary coverage spectrum and permutation spectrum, on the effectiveness of the nearest neighbor model. A binary coverage spectrum contains the coverage of basic blocks whereas a permutation spectrum contains the execution counts of the basic blocks. They evaluate the performance of nearest neighbor models with set union and set intersection models. While, one of the nearest neighbor model uses a binary coverage spectrum, the other nearest neighbor model uses the actual basic block execution counts, i.e. permutation spectrum. They use the Georgia Tech version [23] of the Siemens suite [24] as the subject programs for their experiments. Their work suggests that the nearest neighbor model has a better overall performance when compared to set union and set intersection model. Among the two nearest neighbor models, the one using permutation spectrums has a better effectiveness in locating the fault than the other that uses binary coverage spectrums. Their work is primarily focused on comparing the fault localization effectiveness of nearest neighbor model technique with the set union and set intersection techniques and evaluating the impact of different types of spectra in nearest neighbor's effectiveness. Our work is different from theirs, as we are focused on evaluating the impact of fault properties on the effectiveness of BEN.

The related work discussed so far is focused on evaluating the effectiveness among different fault localization techniques [6], improving the capacity of test cases to improve the effectiveness of fault localization [13], studying the effect of test case reduction on the effectiveness of BEN [14] and evaluating the impact of different spectra on the effectiveness of fault localization technique [15]. To the best of our knowledge,

there are not many empirical studies focused on evaluating the effectiveness of fault localization techniques in locating different fault types.

Bandyopadhyay et al. [12] studied how the effectiveness of a fault localization tool called Tarantula [19] in localizing faults is affected by three fault properties, including accessibility, original state failure condition, and impact. They have used the Siemens suite for their experimental study. Their results indicate that Tarantula, which is also a spectrum based fault localization technique, effectively localizes faults that are hard to access and have low impact.

Our work is different from Bandyopadhyay et al. [12] in the following ways. First, our work focuses on evaluating the effectiveness of BEN, whereas their work is focused on evaluating the effectiveness of Tarantula. Second, while they also evaluate the impact of accessibility on the effectiveness of a fault localization technique, their approach used to measure accessibility is different. In their approach, accessibility is measured by the size of the backward slice of the faulty statement as a percentage of the program size, whereas in our approach accessibility is measured as the ratio of the number of test case executes the faulty statement to the total number of test cases. Third, in addition to the Siemens suite, our experiments were performed on two real-life programs *grep and gzip*. Apart from accessibility, our work evaluates the impact of other two fault properties namely Input Value Sensitivity and Control Flow Sensitivity on the effectiveness of BEN. Using the Input Value Sensitivity property in our study helps understand the impact of co-incidental correctness on the effectiveness of BEN.

CHAPTER 7

CONCLUSION AND FUTURE WORK

The focus of this thesis was on evaluating the effectiveness of BEN in locating different types of fault. In particular, how the three fault properties, namely accessibility, input value sensitivity, and control flow sensitivity affects the effectiveness of BEN in locating a fault. One challenge in our experiments was how to determine the three properties of a given fault. Instead of using an exhaustive test set, which is always nearly impossible in practice, a random test set-based approach was adopted as an approximation to determine the three fault properties. The experimental results suggests, BEN is highly effective in locating faults that are harder to access. This can be explained by the fact that BEN generates a small group of tests whose spectra are compared to locate faults. Experimental results also suggest that in most cases, BEN is effective in locating input value insensitive and control flow-insensitive faults. However, we are unable to reach a conclusion about the individual impact of input value sensitivity and control flow sensitivity on BEN's effectiveness in locating the software fault.

The work presented in this thesis was limited to evaluating BEN's effectiveness in locating different types of fault. Hence, in the future, we plan to extend our work in the following directions. First, we plan to conduct similar studies on other fault localization tools such as Tarantula [19] and Ochiai [17]. Second, the determination of fault properties was performed with manual effort. We plan to automate this process by using scripts that will analyze the GCOV file and random test set results as inputs to determine the properties of the faults automatically. Finally, the subject programs used in the experiments do not have faults that are of high accessibility, input value insensitive and control flow insensitive. Therefore, we were unable to evaluate the impact of such a fault on BEN's effectiveness. We plan to create this type of fault either by using a mutation tool

or by modifying the subject program manually such that the impact of this fault type can be investigated on BEN's effectiveness.

APPENDIX A

CLASSIFICATION OF FAULTS FROM SUBJECT PROGRAMS

Table A-1 - Grep

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
Grep 1	V3	Y	Y	H
	V8	Y	Y	L
	V14	Y	Y	H
Grep 3	V3	Y	Y	H
	V10	Y	Y	H
	V12	N	Y	L
	V16	N	N	L
Grep 4	V2	Y	Y	H
	V12	N	Y	L

Table A-2 - Gzip

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
Gzip1	V2	Y	Y	L
	V4	Y	N	H
	V5	Y	Y	H
	V13	N	Y	L
	V15	N	Y	H
	V16	N	Y	L
Gzip2	V1	N	Y	H
	V3	Y	Y	H
	V6	Y	Y	H
Gzip4	V6	N	Y	L
Gzip5	V1	N	Y	H
	V6	N	Y	L
	V13	N	Y	L

Table A-3 - schedule

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
schedule	v1	Y	Y	H
	v2	Y	N	L
	v3	Y	N	H
	v4	Y	Y	H
	v5	Y	Y	H
	v6	Y	Y	H
	v7	N	Y	L

Table A-4 - schedule2

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
schedule2	v2	Y	Y	H
	v3	Y	Y	H
	v7	Y	Y	H

Table A-5 - totInfo

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
totInfo	v1	Y	Y	L
	v2	Y	Y	H
	v5	Y	N	L
	v6	Y	Y	H
	v8	N	N	L
	v9	Y	N	L
	v11	N	Y	L
	v15	N	Y	L
	v18	Y	Y	H
	v19	Y	Y	H
	v20	Y	Y	H
	v21	Y	Y	H

Table A-6 - printtokens2

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
printtokens2	v1	Y	Y	H
	v2	N	Y	L
	v4	Y	Y	H
	v5	N	Y	L
	v6	Y	Y	H
	v7	Y	Y	H
	v8	Y	Y	H
	v9	Y	Y	H
	v10	Y	Y	H

Table A-7 - printtokens

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
printtokens	v3	N	Y	H
	v5	Y	Y	L
	v6	Y	Y	H

Table A-8 - tcas

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
tcas	v2	Y	Y	H
	v3	Y	Y	H
	v4	Y	Y	L
	v5	Y	Y	H
	v6	Y	Y	L
	v8	Y	Y	H
	v9	Y	Y	H
	v10	Y	Y	L
	v11	Y	Y	L
	v12	Y	Y	H
	v13	Y	Y	H
	v14	Y	Y	H
	v15	Y	Y	H
	v18	Y	Y	H
	v20	Y	Y	H
	v21	Y	Y	H
	v22	Y	Y	H
	v23	Y	Y	H
	v24	Y	Y	H
	v25	Y	Y	L
	v26	Y	Y	H

Table A-8 - continued

tcas	v27	Y	Y	H
	v28	Y	Y	H
	v29	Y	Y	H
	v30	Y	Y	H
	v31	Y	Y	H
	v32	Y	Y	L
	v33	Y	Y	H
	v34	Y	Y	H
	v35	Y	Y	H
	v36	N	N	L
	v37	Y	Y	L
	v38	Y	Y	L
	v39	Y	Y	L
	v40	Y	Y	H
	v41	Y	Y	L

Table A-9 - replace

Subject Program		Input Value Sensitive	Control Flow Sensitive	Accessibility Category
replace	v1	Y	Y	H
	v2	Y	Y	H
	v3	Y	Y	H
	v4	Y	Y	H
	v5	Y	Y	H
	v6	Y	Y	H
	v7	Y	Y	H
	v8	Y	Y	H
	v9	Y	Y	H
	v10	Y	Y	H
	v11	Y	Y	H
	v12	Y	Y	L
	v13	N	Y	L
	v14	Y	Y	L
	v15	Y	Y	H
	v16	Y	Y	H
	v17	N	Y	L
	v18	Y	Y	L
	v19	Y	Y	H
	v20	N	Y	L

Table A - 9 - continued

replace	v21	Y	Y	H
	v22	Y	Y	H
	v23	Y	Y	H
	v24	Y	Y	L
	v25	Y	Y	L
	v26	Y	Y	L
	v27	Y	Y	H
	v28	Y	Y	H
	v29	Y	Y	H
	v30	Y	Y	H
	v31	Y	Y	L
	v32	Y	Y	H

APPENDIX B

EXHAUSTIVE TEST SET - EXECUTION TRACE

Table B-1 - Execution trace for Accessibility Example

Program statements		T1	T2	T3	T4	T5	T6	T7	T8	T9
1.	public int applyDiscount(int totalPrice,bool member, char type)	●	●	●	●	●	●	●	●	●
2.	{	●	●	●	●	●	●	●	●	●
3.	float discount = 0.00;	●	●	●	●	●	●	●	●	●
4.	if(totalPrice>100) //Fault #1 - correct :if(totalPrice>1000)	●	●	●	●	●	●	●	●	●
5.	{	●	●	●				●	●	●
6.	if(member == TRUE)	●	●	●				●	●	●
7.	{	●	●					●	●	
8.	if(type == “E”)	●	●					●	●	
9.	{	●						●		
10.	discount = (0.25)*totalPrice;	●						●		
11.	totalPrice = totalPrice-Discount;	●						●		
12.	}	●						●		
13.	if(type == “G”)	●	●					●	●	
14.	{		●						●	
15.	//Fault # 2 - correct : Discount = (0.10)*totalPrice;									
16.	Discount = (0.07)*totalPrice;		●						●	

Table B - 1 - continued

17.	totalPrice = totalPrice-Discount;		●						●	
18.	}		●						●	
19.	}	●	●					●	●	
20.	else			●						●
21.	{			●						●
22.	discount = (0.05)*totalPrice;			●						●
23.	totalPrice = totalPrice-Discount;			●						●
24.	}			●						●
25.	}	●	●	●						●
26.	}	●	●	●	●	●	●			●

REFERENCES

1. Ghandehari, Laleh Sh, Yu Lei, David Kung, Raghu Kacker, and Ruediger Kuhn. "Fault localization based on failure-inducing combinations." In Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, pp. 168-177. IEEE, 2013.
2. Ghandehari, Laleh Shikh Gholamhossein, Yu Lei, Tao Xie, Richard Kuhn, and Raghu Kacker. "Identifying failure-inducing combinations in a combinatorial test set." In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pp. 370-379. IEEE, 2012.
3. Ghandehari, Laleh Sh, Jaganmohan Chandrasekaran, Yu Lei, Raghu Kacker, and D. Richard Kuhn. "BEN: A combinatorial testing-based fault localization tool." In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pp. 1-4. IEEE, 2015.
4. Hailpern, Brent, and Padmanabhan Santhanam. "Software debugging, testing, and verification." *IBM Systems Journal* 41, no. 1 (2002): 4-12.
5. Wong, W. Eric, and Vidroha Debroy. "A survey of software fault localization." *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45* 9 (2009).
6. Jones, James A., and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique." In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273-282. ACM, 2005.
7. Renieres, Manos, and Steven P. Reiss. "Fault localization with nearest neighbor queries." In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pp. 30-39. IEEE, 2003.

8. Input models for the subject programs., http://barbie.uta.edu/~laleh/BEN/models.html
9. Advanced Combinatorial Testing System (ACTS) http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html
10. L. Ghandehari, Y. Lei, R. Kacker, R, Kuhn. A Combinatorial Testing-Based Approach to Fault localization [Under preparation].
11. Abreu, Rui, Peter Zoeteweij, and Arjan JC Van Gemund. "On the accuracy of spectrum-based fault localization." In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 89-98. IEEE, 2007.
12. Bandyopadhyay, Aritra, and Sudipto Ghosh. "On the Effectiveness of the Tarantula Fault Localization Technique for Different Fault Classes." In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pp. 317-324. IEEE, 2011.
13. Baudry, Benoit, Franck Fleurey, and Yves Le Traon. "Improving test suites for efficient fault localization." In *Proceedings of the 28th international conference on Software engineering*, pp. 82-91. ACM, 2006.
14. Yu, Yanbing, James A. Jones, and Mary Jean Harrold. "An empirical study of the effects of test-suite reduction on fault localization." In *Proceedings of the 30th international conference on Software engineering*, pp. 201-210. ACM, 2008.
15. Renieres, Manos, and Steven P. Reiss. "Fault localization with nearest neighbor queries." In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pp. 30-39. IEEE, 2003.
16. Subject programs - http://sir.unl.edu

17. Abreu, Rui, Peter Zoeteweij, and Arjan JC Van Gemund. "An evaluation of similarity coefficients for software fault localization." In Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on, pp. 39-46. IEEE, 2006.
18. Ghandehari, Laleh Sh, Jacek Czerwonka, Yu Lei, Soheil Shafiee, Raghu Kacker, and Ruediger Kuhn. "An empirical comparison of combinatorial and random testing." In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pp. 68-77. IEEE, 2014.
19. Jones, James A., Mary Jean Harrold, and John Stasko. "Visualization of test information to assist fault localization." In Proceedings of the 24th international conference on Software engineering, pp. 467-477. ACM, 2002.
20. Cleve, Holger, and Andreas Zeller. "Locating causes of program failures." In Proceedings of the 27th international conference on Software engineering, pp. 342-351. ACM, 2005.
21. Chen, Mike Y., Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. "Pinpoint: Problem determination in large, dynamic internet services." In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pp. 595-604. IEEE, 2002.
22. Dallmeier, Valentin, Christian Lindig, and Andreas Zeller. "Lightweight defect localization for java." In ECOOP 2005-Object-Oriented Programming, pp. 528-550. Springer Berlin Heidelberg, 2005.
23. Rothermel, Gregg, and Mary Jean Harrold. "Empirical studies of a safe regression test selection technique." Software Engineering, IEEE Transactions on 24, no. 6 (1998): 401-419.

24. Hutchins, Monica, Herb Foster, Tarak Goradia, and Thomas Ostrand. "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria." In Proceedings of the 16th international conference on Software engineering, pp. 191-200. IEEE Computer Society Press, 1994.
25. Liblit, Ben, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. "Scalable statistical bug isolation." In ACM SIGPLAN Notices, vol. 40, no. 6, pp. 15-26. ACM, 2005.
26. Agrawal, Hiraral, Joseph Horgan, Saul London, and W. Wong. "Fault localization using execution slices and dataflow tests." Proceedings of IEEE Software Reliability Engineering (1995): 143-151.
27. Pan, Hsin, and Eugene H. Spafford. "Heuristics for automatic localization of software faults." World Wide Web (1992).

BIOGRAPHICAL INFORMATION

Jaganmohan Chandrasekaran received his Bachelors of Technology in Information Technology in 2008 from Anna University, Chennai, India. He worked in the industry for 2 years and ten months. He started his Masters in Computer Science at The University of Texas at Arlington in Fall 2013 and joined the Dr.Jeff Lei's group in Spring 2014. His areas of interest are Software Engineering, Software Design Patterns, Automatic Fault Localization, Software Quality and Artificial Intelligence