



Title	Software debugging through dynamic analysis of program structures
Author(s)	Zhang, Zhenyu; 张震宇
Citation	
Issue Date	2010
URL	http://hdl.handle.net/10722/61073
Rights	unrestricted

Abstract of thesis entitled

Software Debugging through Dynamic Analysis of Program Structures

Submitted by

Zhang, Zhenyu

張震宇

for the degree of Doctor of Philosophy

at The University of Hong Kong

in May 2010

Software covers every corner of our lives, so do software faults. Currently, a popular approach in fault localization compares program feature spectra in passed execution and failed executions, and aims at predicting program elements that are close to the faults, by locating program elements whose exercising have strong correlation with program failures. In this thesis, we focus on the so-called statistical fault-localization techniques, investigate four topics, and present the results of our investigations to tackle four related problems.

First, we address the problem that strong correlations may not necessarily be the root cause of the observed failures. We model the propagation traffic through every edge and proportionally apportion the probability of a block being faulty to its directly connected blocks, resulting in a set of homogenous equations in which the probabilities of individual blocks being faulty after propagation of infected program states are the unknown and can be solved. Empirical studies on real-life medium-size programs show that this technique is more effective than state-of-the-art techniques.

Second, we notice that even we can trace the propagation, we may not compute the program feature spectra accurately. On the contrary, distinguishing different short-circuit evaluations that are coarsely written as single program statements may improve existing series of predicate-based fault-localization techniques. We conduct an empirical study which shows that differentiating such short-circuit evaluations of individual program predicates incurs relatively little performance overhead and significantly improves existing such techniques.

Third, in order to identify the locations of the faults, debuggers may require many additional passed executions to be produced after the first failure has been detected; while modern software often provides an error reporting process for



users to feedback the failure information to developers. Our approach opens a new door to perform fault localization solely with failed executions. We use the execution counts of statements to categorize the executions, and calculate the failing rate for each category. We treat every tuple (execution count, failing rate) as a point in two-dimensional space, perform linear regression on them, and eliminate the dependences on passed executions via a signal-to-noise ratio manner. Empirical study shows that this technique is effective.

Fourth, we also focus on the correctness of method used to compare program feature spectra. Many previous studies overlook the statistical distributions of the spectra, on which their parametric techniques fully rely. We have argued and empirically verified that assuming a specific distribution of feature spectra of dynamic program statistics is problematic. We empirically validate that the use of standard non-parametric hypothesis testing methods can achieve better effectiveness than state-of-the-art predicate-based fault-localization techniques.

In conclusion, this thesis contributes to software debugging by developing a family of effective statistical fault-localization techniques. They are particularly useful when (a) the exercising of faulty statements are not strongly corrected to failures, (b) a program has many compound predicates, (c) passed executions are unavailable, or (d) the distribution profile of the program spectra is not available.

[482 words]



Software Debugging
through Dynamic Analysis
of Program Structures

by

Zhang, Zhenyu

張震宇

A thesis submitted in partial fulfillment of the requirements for
the Degree of Doctor of Philosophy
at The University of Hong Kong.

May, 2010



Declaration

I declare that the thesis entitled “Software Debugging through Dynamic Analysis of Program Structures” represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma, or other qualification.

Signed

Zhang, Zhenyu
May 2010



Acknowledgements

My thanks go to my supervisors Prof. T. H. Tse and Dr. W. K. Chan, who give me invaluable support, help, and guidance during my Ph.D. studying period. My thanks also go to Prof. T. Y. Chen, Dr. S. C. Cheung, and Dr. Y. T. Yu, for their help to me on finishing collaborated papers.

I want to thank Bo Jiang of The University of Hong Kong for his hard work on setting up the experiment environment. At the same time, I want to thank Lijun Mei of The University of Hong Kong for his help on proofreading many of my paper drafts. I also need to thank Peifeng Hu for encouraging me in my first year of Ph.D. studying. Peifeng Hu graduated from The University of Hong Kong in the year 2006 and is now working in China Merchant Bank, Hong Kong. I would like to thank Fan Liang of The University of Hong Kong for conducting some validation experiment. My thanks are also given to Xinming Wang of Hong Kong University of Science and Technology for sharing data and experimental tools with me.

My thesis work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111107, 123207, 716507, and 717308) and the Australian Research Council (project number DP0984760).



My Publications

During my Ph.D. studying period, I have the following publications (listed in reverse chronological order).

1. “Fault localization through evaluation sequences” [123], in *Journal of Systems and Software (JSS)* 83(2): 174-187 (2010), coauthored with Bo Jiang, W. K. Chan, T. H. Tse, and Xinming Wang.
2. “Modeling and testing of cloud applications” [29], in *Proceedings of 2009 IEEE Asia-Pacific Services Computing Conference (APSCC 2009)*, coauthored with W. K. Chan, and Lijun Mei.
3. “Adaptive random test case prioritization” [66], in *Proceedings of the 24rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, coauthored with Bo Jiang, W. K. Chan and T. H. Tse.
4. “Capturing propagation of infected program states” [120], in *Proceedings of the 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2009)*, coauthored with W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang.
5. “How well do test case prioritization techniques support statistical fault localization” [67], in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*, coauthored with Bo Jiang, T. H. Tse, and T. Y. Chen. (best paper award)
6. “More tales of clouds: software engineering research issues from the cloud application perspective” [82], short paper, in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*, coauthored with Lijun Mei and W. K. Chan.



7. “Test case prioritization for regression testing of service-oriented business applications” [83], in Proceedings of the 18th International World Wide Web Conference (WWW 2009), coauthored with Lijun Mei, W. K. Chan, and T. H. Tse. (best paper nominee)
8. “Where to adapt dynamic service compositions” [65], poster track, in Proceedings of the 18th International World Wide Web Conference (WWW 2009), coauthored with Bo Jiang, W. K. Chan, and T. H. Tse.
9. “Taming coincidental correctness: coverage refinement with context patterns to improve fault localization” [106], in Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), coauthored with Xinming Wang, S.C. Cheung, and W. K. Chan.
10. “Is non-parametric hypothesis testing model robust for statistical fault localization?” [119], Journal of Information and Software Technology (IST), coauthored with W. K. Chan, T. H. Tse, Peifeng Hu, and Xinming Wang.
11. “Experimental study to compare the use of metamorphic testing and assertion checking” [118], Journal of Software (JoS), coauthored with W. K. Chan, T. H. Tse, and Peifeng Hu.
12. “Fault localization with non-parametric program behavior model” [62], in Proceedings of the 8th International Conference on Quality Software (QSIC 2008), coauthored with Peifeng Hu, W. K. Chan, and T. H. Tse.
13. “Resource prioritization of code optimization techniques for program synthesis of wireless sensor network applications” [121], Journal of Systems and Software (JSS) 82(9): 1376-1387 (2009), coauthored with W. K. Chan, T. H. Tse, Heng Lu, and Lijun Mei. (accepted on Aug 2008)
14. “Debugging through evaluation sequences: a controlled experimental study” [122], in Proceedings of 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008), coauthored with Bo Jiang, W. K. Chan, and T. H. Tse. (best paper award)
15. “Synthesizing component-based WSN applications via automatic combination of code optimization techniques” [117], in Proceedings of the 7th International Conference on Quality Software (QSIC 2007), coauthored with W. K. Chan and T. H. Tse.



16. “Towards the testing of power-aware software applications for wireless sensor networks” [20], in Proceedings of the 12th International Conference on Reliable Software Technologies (Ada-Europe 2007), coauthored with W. K. Chan, T.Y. Chen, S.C. Cheung, and T. H. Tse.
17. “An empirical comparison between direct and indirect test result checking approaches” [61], in Proceedings of the Third International Workshop on Software Quality Assurance (SOQUA 2006), in conjunction with the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering, coauthored with Peifeng Hu, W. K. Chan, and T. H. Tse.





Contents

Declaration	i
Acknowledgements	iii
Contents	viii
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Software Debugging	1
1.2 Fault Localization	1
1.3 Statistical Fault-localization Techniques	2
1.4 Representative Statistical Fault-localization Techniques	3
1.5 Scope of the Thesis	5
1.6 Contributions of the Thesis	8
1.7 Organization of the Thesis	9
2 Literature Review	11
2.1 Program representation	11
2.1.1 Control flow graph	11
2.1.2 Profiling	12
2.2 Fault-localization techniques	12
2.2.1 Statement-level techniques	12
2.2.2 Block-level techniques	13
2.2.3 Branch-level techniques	13
2.2.4 Path-level techniques	14
2.2.5 Multi-level techniques	14
2.2.6 Other kinds of fault-localization work	14



2.3	Fault Repair	15
2.4	Test Case Prioritization in Regression Testing	16
2.5	Oracle Problem in Regression Testing	16
	2.5.1 Assertion checking	17
	2.5.2 Metamorphic testing	17
	2.5.3 Other kinds of methods to alleviate Oracle problem	18
2.6	Summary	19
3	CP: Capturing Propagation of Infected Program States	21
3.1	Background	21
3.2	Motivation	22
3.3	Our Fault-Localization Model	25
	3.3.1 Problem Settings	25
	3.3.2 Preliminaries	26
	3.3.3 Our Model – CP	26
3.4	Experimental Evaluation	34
	3.4.1 Subject Programs	34
	3.4.2 Peer Techniques	35
	3.4.3 Experimental Setup	35
	3.4.4 Effectiveness Metrics	35
	3.4.5 Experiment Environment and Related Issues	36
	3.4.6 Results and Analysis	36
	3.4.7 Threats to Validity	45
3.5	Summary	46
4	DES: Statistical Fault-localization Technique at the Level of Evaluation Sequence	47
4.1	Background	47
4.2	Motivation	48
4.3	Our Fault-localization Model	50
4.4	Research Questions	53
4.5	Experimental Evaluation	54
	4.5.1 Subject Programs	54
	4.5.2 Experimental Setup	55
	4.5.3 Effectiveness Metrics	56
	4.5.4 Results and Analysis	57
	4.5.5 Discussion	67
	4.5.6 Threats to Validity	71
4.6	Summary	72



5	Slope: Statistical Fault Localization via Failure-causing Test Cases Only	73
5.1	Background	73
5.2	Motivation	74
5.3	Our Fault-localization Model	77
5.3.1	Problem Settings	77
5.3.2	Our Observation	77
5.3.3	Our Model – Slope	78
5.3.4	Dealing with Exception Cases	83
5.4	Empirical Evaluation	84
5.4.1	Subject Programs	84
5.4.2	Peer Techniques	84
5.4.3	Experimental Setup	86
5.4.4	Effectiveness Metrics	88
5.4.5	Results and Analysis	88
5.4.6	Threats to Validity	96
5.5	Summary	97
6	Non-parametric Hypothesis Testing Method used in Predicate-based Statistical Fault-localization Techniques	99
6.1	Background	99
6.2	Motivation	102
6.3	Our Fault-localization Framework	106
6.3.1	Preliminaries	106
6.3.2	Problem Settings	107
6.3.3	Our Framework	108
6.4	Research Questions	108
6.5	Experimental Evaluation	111
6.5.1	Subject Programs	111
6.5.2	Peer Techniques and Peer Methods	114
6.5.3	Effectiveness Metrics	114
6.5.4	Experimental Setup	118
6.5.5	Results and Analysis	119
6.5.6	Threats to Validity	139
6.6	Summary	141
7	Further Discussion	143
7.1	Tie-breaking Strategy	143
7.2	Adaptation of our Technique	143



7.3	Fault Fix after Fault Localization	144
7.4	Oracle Problem before Fault Localization	144
7.4.1	Background	144
7.4.2	Research Questions	147
7.4.3	Experiment	148
7.4.4	Summary	164
8	Conclusion	165
	Bibliography	169



List of Figures

1.1	Excerpt from faulty version “v1” of program “tot_info”	4
3.1	Motivating example	23
3.2	Overall effectiveness comparison	38
3.3	Effectiveness on individual programs	41
3.4	Excerpts from multi-fault program	44
4.1	Code excerpts from versions v0 and v8 of print_tokens.	49
4.2	Comparisons of distributions of evaluation biases for evaluation sequences es_1 , es_2 , es_3 , and es_4 (x -axis: evaluation bias; y -axis: no. of test cases).	51
4.3	Comparisons of distributions of evaluation biases for evaluation sequences es_5 and the whole predicate (x -axis: evaluation bias; y -axis: no. of test cases).	52
4.4	Comparisons of DES-enabled techniques with base techniques on all programs.	58
4.5	Comparisons of DES-enabled techniques with base techniques on print_tokens and print_tokens2 programs.	59
4.6	Comparisons of DES-enabled techniques with base techniques on replace program.	59
4.7	Comparisons of DES-enabled techniques with base techniques on schedule and schedule2 programs.	60
4.8	Comparisons of DES-enabled techniques with base techniques on tcas program.	60
4.9	Comparisons of DES-enabled techniques with base techniques on tot_info program.	61
4.10	Comparisons of DES-enabled techniques with base techniques on flex program.	61
4.11	Comparisons of DES-enabled techniques with base techniques on grep program.	62



4.12	Comparisons of DES-enabled techniques with base techniques on gzip program.	62
4.13	Comparisons of DES-enabled techniques with base techniques on sed program.	63
4.14	Code excerpts from versions v0 and v9 of print_tokens2.	67
4.15	Code excerpts from versions v0 and v8 of tot_info.	68
4.16	Code excerpts from versions v0 and v8 of tot_info.	68
5.1	Faulty version v10 of replace	75
5.2	Failing rate vs. execution count	75
5.3	Framework of our model	79
5.4	Overall results on all faulty versions	89
5.5	Effectiveness difference on all faulty versions	92
5.6	Faulty version v28 of tcas	96
6.1	Excerpt from faulty version “v1” of program “tot_info”	103
6.2	Distributions of evaluation biases for predicates P_1 to P_4	104
6.3	Distributions of evaluation biases for predicates P_5 to P_7	105
6.4	Illustration for normality test	117
6.5	Illustration of Pearson Correlation test	118
6.6	Overall effectiveness comparisons	120
6.7	Effect of test suite size (lower the curve, better the technique)	121
6.8	Time spend of each techniques on different programs	123
6.9	Effectiveness comparisons on print_tokens & print_tokens2	124
6.10	Effectiveness comparisons on replace	125
6.11	Effectiveness comparisons on schedule & schedule2	125
6.12	Effectiveness comparisons on tcas	126
6.13	Effectiveness comparisons on tot_info	126
6.14	Results of normality test for predicates	130
6.15	Results of normality test for the most fault-relevant predicates	134
6.16	Effect of normality on fault-localization techniques	138
7.1	Experiences of subjects in object-oriented design, Java, testing, and assertions	151
7.2	Box-and-whisker plots of time costs for applying MT and assertion checking	158



List of Tables

3.1	Statistics of subject programs	34
3.2	Statistics of effectiveness	42
3.3	Statistics of differences in effectiveness	43
4.1	Evaluation sequences of code fragments.	49
4.2	Statistics of subject programs.	56
4.3	Statistics on relative improvements in effectiveness.	64
4.4	p-values of U-tests on Siemens programs and UNIX programs.	65
4.5	Timing statistics in the experiment.	71
5.1	Statistics of subject programs	85
5.2	Statistics of individual results on 186 single-fault versions	93
5.3	Statistics of individual results on 20 multi-fault versions	93
5.4	Results on Pearson correlation test to compare the effectiveness of techniques	94
5.5	Results of t-test hypothesis testing to compare the effectiveness of techniques	95
6.1	Techniques we are interested in	109
6.2	Statistics of Siemens suite	112
6.3	Important fault types for C programs	113
6.4	The p-value results of hypothesis “technique X has significant improvements on technique Y ”	128
6.5	Student’s t-test on different threshold for $H1$	132
6.6	Comparison of statistics of predicates with statistics of the most fault-relevant predicates	135
6.7	Student’s t-test on different threshold for $H2$	136
7.1	Statistics of target programs	149
7.2	Categories of mutation operators	150
7.3	Number of single-fault programs by mutation operator category	152



7.4	Number of metamorphic relations and assertions	154
7.5	Normalized standard derivations	154
7.6	Mutation detection ratios for metamorphic testing and assertion checking	156
7.7	Statistics of time costs for applying MT and assertion checking .	157
7.8	Mutation detection ratios for MR with and without faulty metamorphic relation implementation	159
7.9	Examples of metamorphic relations for program Boyer	161



Dedicated to my family





Chapter 1

Introduction

This chapter introduces the concepts and conventions in software debugging, fault localization, and statistical fault localization, and then presents the scope of this thesis. After that, this chapter summarizes the main contributions of the thesis and highlights its organization for ease of reference.

1.1 Software Debugging

While software develops and evolves fast, software failures are hard to avoid. Software faults are due to mistakes made during the implementation phase. Software debugging is the process of removing faults in programs. While it is a crucial activity, it is time-consuming and takes up a substantial portion of the effort [103] as well as a significant amount of the resources [68] in a typical software project. During software testing, programs may exhibit abnormal behaviors known as failures. To eliminate such abnormal behaviors, debuggers need to locate the fault, then repair the fault and perform regression testing to verify the result [103]. Compared with fault repair and regression testing, fault localization is the most important task in software debugging [68].

1.2 Fault Localization

Fault localization is recognized as the hardest, most tedious, and most time-consuming task in software debugging [103]. Generally speaking, *fault localization* is to identify faults that cause abnormal behaviors in a program. Using an effective fault-localization technique to assist debuggers to find bugs is a long-standing means to alleviate the problem. Traditionally, debuggers iteratively and



repeatedly set breakpoints, insert assertions, execute the program over inputs, monitor the program states, and identify suspicious program elements [62].

1.3 Statistical Fault-localization Techniques

We first give a few preliminaries to support statistical fault localization. In this thesis, we use a *failed execution* to denote a program execution that reveals a failure. We use a *passed execution* to denote a program execution that reveals no failure. A *passed test case* is one that shows no failure, and a *failure-causing test case* is one identified to have detected a failure [120].

To escape from the previous time-consuming and error-prone manual fault-localization process, automatic fault-localization techniques [68][70][74][75][76][77] have been invented. A commonly used tactic in such techniques is to apply a statistical approach to correlate program failures with program entities (such as statements or predicates). They compare the program feature spectra (such as coverage information of statements or predicates) in failed executions and passed executions, and find suspicious program elements to facilitate the identification of faults. Since such an approach uses dynamic program execution information, the corresponding techniques are categorized as “dynamic analysis” [120]. (The counterpart is “static analysis”, which analyzes programs statically.) These techniques narrow down on suspicious regions of fault-relevant program elements by finding those elements whose executions correlate with program failures. For example, ideally, if a statement is always exercised in failed executions and never exercised in passed executions, it has a high probability to be related to fault. In previous work, such techniques have been empirically validated to be promising [76][77].

A key insight is based on the assumption that certain dynamic *feature* of program entities is more sensitive to the *difference* between the set of failed executions and the set of all (or passed) executions. Thus, there are two key elements underlying the successful applications of such class of dynamic analysis techniques. First, a technique should use a feature (or a set of features) to measure the sensitivity (that is, how much a program element is estimated to be related to faults in a program). Second, the technique should have a function to compare sensitivity values. The function essentially ranks sensitivity values in a total order. For example, techniques such as [75][76] produce a real number value to represent sensitivity, and sort these values in ascending or descending order. By mapping the relative order of the sensitivity values back to the associated program entities, such a technique can produce a ranked list of program entities



accordingly. We note that the relative magnitudes of sensitivity values are used when ranking the program entities. Note also that the relative magnitudes of sensitivity values rather than their absolute values are used because the value ranges can be unbounded in general [120]. Since these kinds of fault-localization techniques often make use of statistical methods, e.g., correlation test and hypothesis test, to investigate statistical information (program execution information over a suite of test cases), we also call them “*statistical fault-localization techniques*”.

1.4 Representative Statistical Fault-localization Techniques

Control flow graphs [6] have been designed to model program structures. In a control flow graph, the branch transitions (happening, say, at an “if”-switch) of program executions are represented by edges, while the statements between two adjacent edges and sequentially executed as a whole or skipped as a whole (such as the statements in the true branch of an “if”-statement) are represented by blocks. Control flow blocks are connected to one another by control flow edges. Such a control flow graph can be used to capture dynamic information of program execution, e.g., the execution path of a program over a given test case. Many common statistical fault-localization techniques [4][35][68][96][106][112] can be deemed to use control flow elements at different levels to perform fault localization.

Some existing popular techniques use statements as fault location indicators in order to investigate the fault-suspiciousness of every statement, i.e., how much it is related to a fault. They regard statements as program entities, count the number of times a program entity is exercised in a program execution, and use such execution counts as dynamic features. We will refer to them as statement-level statistical fault-localization techniques. Tarantula [68], Jaccard [1], and Ochiai [1] are examples of such techniques.

Branch-level statistical fault-localization techniques [96] considers branches as program entities, and uses the frequencies of branch (i.e., how many times branches are exercised in a program execution) as dynamic features. One example of such techniques is “br” in [96].

At the same time, many previous studies [75][76][77] have found that the frequencies of branch transition through a control flow edge correlate to how much that edge is related to faults in a program. One strategy is to use the transition through control flow edges (e.g., branch statements and return statements) as predicates, then use such predicates as program entities, and the execution



Program : tot_info.c [62]	
P_1 :	if (rdf \leq 0 cdf \leq 0) { info = -3.0; goto ret3; }
	⋮
P_2 :	for (i = 0; i < r; ++i) { double sum = 0.0;
P_3 :	for (j = 0; j < c; ++j) {
P_4 :	long k = x(i,j);
E_1 :	if (k < 0L) { info = -2.0; /*goto ret1;*/
	} sum += (double)k;
	} N += xi[i] = sum;
P_5 :	if (N \leq 0.0) { info = -1.0; goto ret1; }
P_6 :	for (j = 0; j < c; ++j) { double sum = 0.0;
P_7 :	for (i = 0; i < r; ++i) sum += (double)x(i,j); xj[j] = sum;
	}
	⋮
	ret1:

Figure 1.1: Excerpt from faulty version “v1” of program “tot_info”

counts and execution results as the dynamic features. Here, a *program predicate* (or simply *predicate*) about some property of execution at a program location may be evaluated to a certain truth value. For example, we may regard the condition “ $\text{rdf} \leq 0 \parallel \text{cdf} \leq 0$ ” of the branch statement “ $P_1: \text{if}(\text{rdf} \leq 0 \parallel \text{cdf} \leq 0)$ ” in Figure 1.1 as a predicate. If this branch statement is exercised in the program execution and the condition is evaluated to be *true* (e.g., the value of variable “rdf” is less than zero), we say the corresponding predicate is evaluated to be *true* with respect to that execution. If this branch statement is exercised in the program execution and the condition is evaluated to be *false*, we say the corresponding predicate is evaluated to be *false* with respect to that execution. If this branch statement is not exercised and the condition is not evaluated, we say the predicate is not evaluated with respect to that execution. Some existing statistical fault-localization techniques [62][74][75][76][77][119][122][123] locate suspicious program predicates related to faults by contrasting the behaviors of the program predicates (e.g., decisions of branch statements) in passed executions and failed executions.

Based on these kinds of predicate-level techniques, path-level techniques [35] have been developed. Such techniques isolate bugs by finding paths that correlate with failure. Their results indicate that “*path profiles can help isolate bugs more precisely by providing more information about the context in which bugs occur*” [35].

1.5 Scope of the Thesis

In this thesis, we present investigations of four important topics in statistical fault-localization.

1. We notice that an infected program state triggered by a fault may propagate a long way before the program execution finally causes a failure. The execution of such a faulty statement has a relatively weak correlation with program failures. Since many existing statistical fault-localization techniques locate program positions whose executions correlate directly with program failures, they are not effective to locate such faults. To address this kind of fault, we present in this thesis a technique known as CP [120], which captures the propagation of infected program states. CP also uses a control flow graph (CFG) to describe a given program. It models a program execution as an edge profile [5], which indicates the edges of the CFG that have been traversed during the execution, and quantifies



every change of program state over an edge with the number of traversals of the edge. It then computes a pair of mean edge profiles [120]: a mean pass profile for all the sampled passed executions, and a mean failed profile for all the sampled failed executions. They abstractly capture the central tendency of the program states in a passed execution and that in a failed execution, respectively. For each edge, CP contrasts such a state abstraction in the mean pass profile with that in the mean failed profile to assess the fault suspiciousness of the edge. To track how much every basic block [5] contributes to the observed program failures, CP sets up a system of linear algebraic equations to express the propagation of the suspiciousness scores of a basic block to its predecessor block(s) via directly connected control flow edges — for each edge, it splits a fraction of the suspiciousness score to be propagated to a predecessor basic block. CP always constructs homogeneous equations and ensures that the number of equations is the same as the number of variables. Such a constructed equation set is easily solvable by standard mathematics techniques such as Gaussian elimination and least square method [125]. By solving the set of equations, CP obtains the suspiciousness score for each basic block. It finally ranks the basic blocks in descending order of their suspiciousness scores, and assigns a rank for each statement accordingly.

2. During experimental evaluation, we notice that peer predicate-level fault-localization techniques are not as effective as other statement-level techniques. We then continue to investigate the factors that have impact on the effectiveness of predicate-level fault-localization techniques. These techniques find fault-relevant predicates in a program by contrasting the statistics of the evaluation results for individual predicates between passed executions and failed executions. However, short-circuit evaluations may occur when evaluating compound Boolean expressions in program executions. Treating predicates as atomic units ignores this fact, masking out various types of useful statistics on dynamic program behavior. We therefore develop a finer-grained technique, known as DES [122][123], that considers the impact of short-circuit evaluations. We use the concept of evaluation sequence to capture the details of evaluating a compound Boolean expression, and collect the evaluation biases in terms of evaluation sequences. Our intent is to investigate the performance overhead and effectiveness improvements of such a finer-grained statistical fault-localization technique.
3. We further present a novel statement-level statistical fault-localization tech-



nique, Slope, which can work in the absence of passed executions. Such a technique greatly save the effort of generating or exercising passed test cases. It also enables fault localization in environments where passed executions are not ready or unavailable, such as a debugging request driven by bug reports. In detail, executing a faulty statement (such as a statement having a boundary-checking fault) *once* may not sufficiently lead to a failure. However, we have the following observation based on real-life debugging experience. “The more frequently a fault has been exercised in an execution, the more likely will the fault affect the program behavior and, thus, the higher will be the chance that the program execution results in a failure.” Based on this experience, we propose our technique, known as Slope, to address this problem. It uses the trend of chances on each applicable statement to estimate the fault suspiciousness of that statement. Slope is summarized as follows: It collects the execution count of every statement in each execution, where the execution count of a statement is the number of times that the statement has been exercised in an execution. It then uses the execution count as the criterion to categorize the executions into subgroups. Slope calculates the failing rate in each subgroup, and treats every tuple failing rate, execution count as a point in two-dimensional space. It lines up these points using linear regression, and treats the slope of the obtained line as the desirable signal and fitting error as the noise of the signal to construct the ranking formula by means of a signal-to-noise ratio, which is also known as the inverse of coefficient of variation [95]. It makes use of the dimensionless property of the signal-to-noise ratio, with the help of further approximation, to make our model independent of passed executions. Based on the model, we develop a novel statistical fault-localization technique, which has two ranking formulas sharing the same underlying principle but applicable to two different scenarios (that is, the “pass and fail” scenario and the “fail only” scenario). The former relates to traditional statistical fault-localization environment where both passed and failed executions are available. The latter stands for a fault-localization environment where only failed executions are available.

4. Generally, to apply their models, many statistical fault-localization techniques [74][75][76][77] set up presumptions that feature spectra exhibit specific distributions. Using an unmatched model to describe the realistic feature spectra may result in low effectiveness for these techniques. We use standard non-parametric hypothesis testing methods instead. In our



previous work [60, 62], we have empirically shown that the use of a non-parametric hypothesis testing method can improve the effectiveness of existing techniques. In this thesis, we would like to find the root cause of improvement by contrasting the effectiveness of fault-localization techniques using different non-parametric or parametric hypothesis testing methods with the effectiveness of existing self-proposed parametric hypothesis testing models.

We have conducted controlled experiments to validate the effectiveness of the above four kinds of fault-localization techniques. The empirical results show that both Slope and CP are effective. On the other hand, the finer-grained (at level of evaluation sequence) statistical fault-localization technique DES significantly improves existing techniques while incurring relatively little overhead. Moreover, a statistical fault-localization technique using standard non-parametric hypothesis testing methods is extremely valid and robust.

Since a test suite is part of the input of statistical fault localization, in the follow-up discussion section, we further investigate and compare the effectiveness of the use of metamorphic testing [60] and assertion checking [60] to alleviate the oracle problem [33] in software testing.

1.6 Contributions of the Thesis

The following summarizes the major contributions of the thesis. Further details will be given in the relevant chapters that follow.

1. We propose two novel statistical fault-localization techniques, which are evaluated as effective by controlled experiments using realistic medium-sized real-life subject programs and representative peer techniques for comparison.
2. We conduct the first study to investigate the impact of short-circuit evaluation of compound Boolean expressions on the effectiveness of statistical fault-localization techniques.
3. We present the first investigations on the normality nature of the program execution spectra and the short-circuit evaluation rule, which highlight some threats to the construct validity of existing predicate-level fault-localization techniques.
4. We propose a new metrics, P-score, to measure the effectiveness of predicate-level fault-localization techniques.



5. We conduct the first controlled experiment to compare metamorphic testing and assertion checking, which are two methods used to alleviate the oracle problem in software testing. It also empirically provides a tradeoff guide when choosing between metamorphic testing and assertion checking.

1.7 Organization of the Thesis

This thesis is organized as follows: In Chapter 2, we discuss related work and give a literature review. In Chapter 3, we propose a steady and efficient propagation-based statistical fault-localization technique known as CP. In Chapter 4, we investigate whether a finer-grained concept — evaluation sequences — of predicates is significant for improving the effectiveness of predicate-based statistical fault localization. In Chapter 5, we introduce our fault-localization work, Slope, to explain how we may solely use failed executions to locate fault. In Chapter 6, we extend our previous work by applying different non-parametric hypothesis testing methods and standard parametric hypothesis testing methods to our fault-localization framework in a controlled experiment. In Chapter 7, we discuss related issues and report the results of a controlled experiment that compares two methods used to alleviate the oracle problem. Finally, Chapter 8 concludes the thesis.



Chapter 2

Literature Review

This chapter lists work related to this thesis and discusses their relations with it.

2.1 Program representation

2.1.1 Control flow graph

Control flow graph [6] is designed to describe program structure. In a control flow graph, the branch transitions (e.g., a branch transition happens at an “if”-switch) of program execution are represented as edges, the statements between two adjacent edges and sequentially executed as a whole or skipped as a whole (e.g., the statements in the true block of an “if”-statement) are represented as basic blocks. Control flow blocks are connected to one another by control flow edges.

Given program M , a control flow graph (CFG) [6] can be denoted as,

$$G(M) = \langle E, B \rangle.$$

Here $E = \{e_1, e_2, \dots, e_m\}$ is a set of the control flow edges of M , and $B = \{b_1, b_2, \dots, b_n\}$ is a set of the basic blocks of M .

Such a control flow graph can be used to analyze the program behavior or structure [120]. It can be also used to capture dynamic information of program execution (e.g., the execution path with respect to the program execution over a given test case) and a lot of common statistical fault-localization techniques [4][35][68][96][106][112] can be regarded to use control flow elements of different levels to perform fault localization.



2.1.2 Profiling

Profiling techniques are useful to collect structured execution information of program, such as a control flow graph. It can be used to support instrumentation work in fault-localization techniques [120]. Profiling techniques have been developed for years. For example, Bond et al. [15] propose hybrid instrumentation and sampling approach for continuous path and edge profiling. Ball et al. in their work [7] compare edge profiling and path profiling and conclude that edge profiling is in most cases enough to select hot path instead of using path profiling.

2.2 Fault-localization techniques

Many various techniques [4][35][68][96][106][109][112] have been proposed to support software debugging. For example, Wong et al. [109] propose a code coverage-based fault-localization technique, which uses a utility function to calibrate the contribution of each passed execution when calculating the fault relevance of executed statements. These techniques usually contrast the program spectra information [64] (such as execution statistics) between passed executions and failed executions to compute the fault suspiciousness [112] of individual program entities (such as statements [68], branches [96], and predicates [75]), and construct a list of program entities in descending order of their fault suspiciousness. Developers may then follow the suggested list to locate faults. Empirical studies [4][68][75][76] show that these techniques can be effective in guiding programmers to examine code and locate faults.

2.2.1 Statement-level techniques

Harrold et al. [57] list nine classes of program spectra, such as path count, data-dependency count, and execution trace. Among them, the execution trace spectrum is most widely used in debugging. *Tarantula* [68] is a statement-level statistical fault-localization technique. In their work *Tarantula*, Jones et al. [68][70] rank each statement according to suspiciousness, which is a function of the percentages of failed and passed test cases that execute the statement. They further use *Tarantula* to explore ways of classifying test cases to enable several test engineers to debug a faulty program in parallel [69]. There are many other statement-level fault-localization techniques, for example, *Jaccard* [1] and *Ochiai* [1]. The difference among them and *Tarantula* is that they use different ranking formulas



and do not have tie-break strategy. Yu et al. [112] further summarize several other statement-based fault-localization techniques.

2.2.2 Block-level techniques

When statements have identical execution statistics and *Tarantula* cannot distinguish them, it involves additional strategy to break ties [112]. For example, blocks contain statements that will be sequentially executed as a whole or skipped as a whole. These statements always have the same execution count during program executions. They are statistically indistinguishable since these kinds of techniques rely on execution counts to determine the fault-suspiciousness of statements. We name such techniques block-level statistical fault-localization techniques. Baudry et al. [8] define a *dynamic basic block* as the set of statements executed by the same test cases in a test suite. They use a bacteriologic approach to remove test cases while maximizing the number of dynamic basic blocks, and use the algorithm in [70] to rank the statements. They manage to use fewer test cases than *Tarantula* for the same fault-localization results.

2.2.3 Branch-level techniques

At the same time, many previous studies [75][76][77] have found that the frequencies of branch transition through a control flow edge correlate to how much that edge is related to faults in program. One strategy is to use the transition through control flow edges (e.g., branch statements and return statements) as predicates, use such predicates as program entities, and the execution counts and execution results as the dynamic features. If a branch statement is exercised in the program execution and this condition is evaluated to be *true*, we say the corresponding predicate is evaluated to be *true*, with respect to that execution. If this branch statement is exercised in the program execution and this condition is evaluated to be *false*, we say the corresponding predicate is evaluated to be *false*, with respect to that execution. If this branch statement is not exercised and this condition is not evaluated, we say the predicate is not evaluated, with respect to that execution. Some previous statistical fault-localization techniques [62][74][75][76][77] locate the suspicious program predicates that relate to faults by contrasting the behavior of program predicates (e.g., decisions of branch statements), in passed executions and failed executions. The *SOBER* approach [76] proposes to contrast the differences between a set of evaluation biases due to passed test cases and that due to failure-causing ones for every predicate in the program. It hypothesizes that, the greater is the difference be-



tween such a pair of sets of evaluation biases, the higher will be the chance that the corresponding predicate is fault-relevant. Liblit et al. [75] propose a sparse sampling approach *CBI* to collect the statistics of predicates for statistical fault localization. By sampling selected predicates, rather than all predicates or statements, this strategy reduces the overhead in collecting debugging information. It also reduces the need to disclose the execution details of all statements when remote sampling is conducted (for the purpose of remote support rather than on-site support [74][75]). Hence, it lowers the risk of information leakage, which is a security concern. The *CBI* approach [74][75] proposes a heuristic that measures the increase in probability that a predicate is evaluated to be *true* in a set of failure-causing test cases, compared to the whole set of (passed and failure-causing) test cases. They further adapt *CBI* to exploit the execution statistics of compound Boolean expressions constructed from program predicates to facilitate statistical debugging [4]. *CBI* is also adapted to the statement level in previous work [112].

2.2.4 Path-level techniques

Based on these kinds of predicate-level techniques, path-level techniques [35][38] are developed. Chilimbi et al. [35] extend *CBI* by using path profiling to locate faults. Such a technique isolates bugs by finding paths that correlate with failure. Previous results indicate that “*path profiles can help isolate bugs more precisely by providing more information about the context in which bugs occur*” [35].

2.2.5 Multi-level techniques

Similar to and inspired by statement-level techniques, Santelices et al. [96] investigate the effectiveness of using different program entities (statements, edges, and DU-pairs) to locate faults. They show that the integrated results of using different program entities may be better than the use of any single kind of program entity.

2.2.6 Other kinds of fault-localization work

Besides the above coverage-based fault-localization techniques [106], there are also fault-localization techniques, such as [64], that make use of both coverage information and detailed data flow information. They collect all possible values of program variables, iteratively replace the value of each program variable, calculate the probability that such a replacement converts a failure-causing test case



to a passed test case, and estimate the suspiciousness of statements accordingly.

Delta Debugging [36][114] isolates failure-inducing input elements, produces cause-effect chains, and locates the faults through the analysis of program state changes during a failed execution against a passed one. It isolates the relevant variables and values by systematically reducing the state differences between a failed execution and a passed one.

Renieris and Reiss [90] find the difference in execution traces between a failed execution and its *Nearest Neighbor* passed execution to be effective for debugging. Statements with unsymmetrical differences between failed and passed executions are regarded as faulty statements. Another contribution of this work is its proposed effectiveness metrics *t-score*, which is also used in this thesis to evaluate the effectiveness of some fault-localization techniques.

Program slicing [107] is a code-based technique. It is widely used in debugging [100][116]. Gupta et al. [56] propose a forward dynamic slicing approach to narrow down slices. They further integrate the forward approach with standard dynamic slicing approaches [115].

Model checking is a verification method that accepts temporal logic formulas or other modeling languages. Griesmayer et al. [54] use model checking to locate faults. Beginning with given specification, a model checker delivers a counterexample, which reports the suspicious program states and helps locate fault. By searching the error traces, expressions that repair the original program can be constructed.

Some research studies such as failure report analysis [88][104] classify failures based on failed executions. However, they do not address the problem of solely using failed executions to locate faults (which is part of our contribution in this thesis). Besides, Ko and Myers in their work [71] discuss another manner of fault localization rather than outputting a suspicious statement list [72]. Locating multiple faults simultaneously is also a research direction [124].

2.3 Fault Repair

There also exist automatic fault repairing techniques. For example, Sinha et al. [96] “*handle runtime exceptions that involve a flow of an incorrect value that finally leads to the exception*” [96]. It traces back from a statement where an exception occurred, and then combines dynamic information and static data-flow analysis to identify the fault. It also identifies those statements causing the same exception in program execution of other test cases, and finally makes use of such information to repair the fault.



Predicate switching [115] alters a predicate's decision at execution time to change the original control flow of a faulty program over the failure-causing test case, aiming to find a key predicate that triggers the faulty program to give correct output. It then searches from this switched predicate to locate a fault through backward and forward slicing. Making use of slicing technique, Jeffrey et al. propose *Value Replacement* [64] for fault localization. It collects all possible values from all executions of a test suite, and then replaces the value of each variable occurrence in the faulty program by each of these collected values systematically to determine whether such a replacement can produce the correct output. Since Value Replacement uses both coverage information and variable values, we do not regard it as a peer technique for comparison. These two studies can be deemed to repair a special kind of faults.

2.4 Test Case Prioritization in Regression Testing

Our work [67] also investigates how test case prioritization techniques have impact on the effectiveness of fault localization. In regression test phase, test case prioritization techniques [41][43][46][48][49][50][51][52][92][93][94] are commonly used to prioritize the test cases and gain a fast speed of revealing failures.

Do and Rothermel in their work [45] use sensitivity analysis to determine the dominant factor of their model, and propose two new models based on the result. We assess these models empirically on data obtained by using regression testing techniques on several non-trivial software systems. However, such work is out of the scope of this thesis. Do et al. [42][44] investigate the use of mutation faults in regression testing.

In our work [67], we investigate the impact of those test case prioritization techniques (used in regression testing) on the effect of fault-localization techniques (used in fault localization), and report the practical issue of using these test case prioritization techniques (fault localization in a continuous integration environment).

2.5 Oracle Problem in Regression Testing

Oracle problem affects the quality of test suite, which is the input of statistical fault localization. Many approaches have been proposed to alleviate the test oracle problem. Rather than checking the test output directly, they usually propose to construct various types of oracle variant to verify the correctness of the program under test. Chapman [30] suggested that a previous version of a program



could be used to verify the correctness of the current version. It is now a popular practice in regression testing. However, using this approach, testers need to identify whether the test case is applicable to the previous version.

2.5.1 Assertion checking

Assertion checking [87] is a method to verify the execution results of programs. An assertion, which is usually embedded directly in the source code of the program under test, is a Boolean expression that verifies whether the execution of a test case satisfies some necessary properties for correct implementation. Assertions are supported by many programming languages and are easy to implement. It has been incorporated in the Microsoft .Net platform.

Assertion checking has been widely used in testing. For example, state invariants [11][58], represented by assertions, can be used to check the stated-based behaviors of a system. Briand et al. [17] investigate the effectiveness of using state-invariant assertions as oracles and compared it with the results using precise oracles for object-oriented programs. It is shown that state-invariant assertions are effective in detecting state-related errors. Since our target programs are also object-oriented programs, we have chosen assertion checking as the alternative testing strategy in our experimental comparison. Assertion checking is also popular in unit testing framework such as JUnit, in which verification of the program states or outputs of a test case can be done during or after the test execution.

2.5.2 Metamorphic testing

There have been various case studies in applying *Metamorphic Testing* [20][21][24][25][31][33][55][62][101] to different types of programs, ranging from conventional programs and object-oriented programs, to pervasive programs and web services. Chen et al. [31] report on the testing of programs for solving partial differential equations. They [32] further investigate the integration of metamorphic testing with fault-based testing and global symbolic evaluation. Gotlieb and Botella [55] develop an automated framework to check against a class of metamorphic relations. In previous studies, metamorphic approach is also applied to unit testing [101] and integration testing [21] of context-sensitive middleware-based applications. Chan and others [24][25] also develop a metamorphic approach to online testing of service-oriented software applications. The improvement on the binary classification approach to alleviate the test oracle problem for graphics-intensive applications has been investigated in [26][27][28].



Beydeda [10] proposes to use metamorphic testing as a means to improve the testability of program components. Wu [110] observes that follow-up test cases can be initial test cases of the next round, and thus, proposes to apply MT iteratively to utilize metamorphic relations more economically. Chan et al. [27][28] propose a methodology to integrate MT with the pattern classification technique. Murphy [85] explores the application of metamorphic testing to support field testing. Throughout these studies, both the testing and the evaluation of experimental results were conducted by the researchers themselves. There is a need for systematic empirical research on how well MT can be applied in practical and yet generic situations and how effective MT is compared with other testing strategies.

2.5.3 Other kinds of methods to alleviate Oracle problem

Some researchers have proposed to prepare test specifications, either manually or automatically, to alleviate the test oracle problem. Memon et al. [84] assume that a test specification of internal object interactions was available and used it to identify non-conformance of the execution traces. This type of approach is common in conformance testing for telecommunication protocols. Sun et al. [98] propose a similar approach to testing the harnesses of applications. Last and colleagues [73][102] train pattern classifiers to learn the casual input-output relationships of a legacy system. They then use the classifiers as test oracles. Chan et al. [26] further investigate the feasibility of using pattern classification techniques when the test outputs cannot be accurately determined. Podgurski and colleagues [53][88] classify failure reports into categories via classifiers, and then refine the classification with the aim to extract more information to help testers diagnose program failures. Bowring et al. [16] use a progressive approach to train a classifier to ease the test oracle problem in regression testing. Chan et al. [23] use classifiers to identify different types of behaviors related to the synchronization failures of objects in a multimedia application.

Weyuker [108] suggests checking whether some identity relations would be preserved by the program under test. This notion of equivalence has been well-adopted in practice. Blum and others [3][12] propose a program checker, which is an algorithm for checking the output of computation for numerical programs. Their theory has been subsequently extended into the theory of self-testing/correcting [13]. Xie and Memon [111] study different types of oracle for graphic user interface (GUI) testing. Binder [11] discusses four categories and eighteen oracle patterns in object-oriented program testing.



2.6 Summary

In this chapter, we list out much work related to this thesis, including fault-localization techniques, fault repair, oracle problem, test case prioritization, control flow graph, and profiling techniques.

We also investigate program synthesizer for resource-stringent *WSN* applications [117][121]. Based on different program structures, they use heuristic to search for proper combination of code optimization techniques [89], to meet resource criteria.

Besides, our work [66] investigates the use of adaptive random testing in regression testing. Another work [83] investigates the test case prioritization techniques applied on Service-Oriented Business Applications.



Chapter 3

Capturing Propagation of Infected Program States

We notice that many previous statement-level statistical fault-localization techniques work by locating program statements which execution correlate to program failures. Since an infected program state triggered by a faulty statement may propagate during program execution, and goes a long way before a failure is finally caused. The execution of such a faulty statement does not strongly correlate to program failures and previous statement-level fault-localization techniques are not effective on locating it.

In this chapter, we first give some necessary background, and then use a motivating example to demonstrate such a case and how we address it. After that, we elaborate on our model – CP [120], which captures the propagation of infected program states to locate faults, and then use empirical study to validate the effectiveness of our technique.

3.1 Background

During program execution, a fault in a program statement may infect a program state, and yet the execution may further propagate the infected program states [36][104] a long way before it may finally manifest failures [106]. Moreover, even if every failed execution may execute a particular statement, this statement is not necessarily the root cause of the failure (that is, the fault that directly leads to the failure) [36].

Suppose, for instance that a particular statement S on a branch B always sets up a null pointer variable. Suppose further that this pointer variable will not be used in any execution to invoke any function, until another faraway (in the sense of control dependence [6] or data dependence) statement S' on a branch B' has been reached, which will crash the program. If S is also exercised in many other executions that do not show any failure, S or its



directly connected branches cannot effectively be pinpointed as suspicious. In this scenario, existing statistical fault-localization techniques such as Tarantula [68] or SBI [112] will rank S' as more suspicious than S . Indeed, in the above scenario, exercising B' that determines the execution of S' always leads to a failure [112]. Thus, the branch technique proposed in [112], for example, will rank B' as more suspicious than B , which in fact is directly connected to the first statement S . The use of data flow analysis may reveal the usage of the null pointer and help evaluate the suspiciousness of S , S' , B , and B' . However, data flow profiling is expensive [64][96].

A way out is to abstract a concrete program state as a control flow branch, and abstract the propagation of fault suspiciousness of these concrete program states by a “transfer function” of the fault suspiciousness of one branch or statement to other branches or statements. On one hand, existing techniques work at the individual program entity level and assess the fault suspiciousness of program entities separately. On the other hand, the transfer of fault suspiciousness of one program entity to another will change the fault suspiciousness of the latter. In the presence of loops, finding a stable propagation is non-trivial. Moreover, even if a stable propagation can be found, a direct implementation of such a propagation-based technique may indicate that the technique requires many rounds of iterations, which unfortunately are computationally expensive.

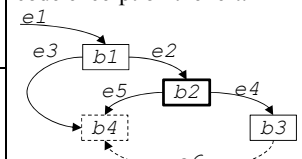
3.2 Motivation

Figure 3.1 shows a code excerpt from the faulty version v2 of the program schedule (from SIR [40]). The code excerpt manages a process queue. It first calculates the index of a target process, and then moves the target process among priority queues. We seed an extra “+1” operation fault into statement s_2 in Figure 3.1. It may cause the program to select an incorrect operation for subsequent processing, which will lead to a failure.

This code excerpt contains two “if” statements (s_1 and s_5), which divide the code excerpt into three basic blocks Figure 3.1 (namely, b_1 , b_2 , and b_3). The basic block b_1 contains only one statement s_1 . The result of evaluating “block_queue” in s_1 determines whether the statements s_2 to s_8 are skipped. The basic block b_2 contains statements s_2 , s_3 , s_4 , and s_5 . The basic block b_3 contains statements s_6 , s_7 , and s_8 . We also depict the code excerpt as a control flow graph (CFG) in Figure 3.1. In this CFG, each rectangular box represents a basic block and each arrow represents a control flow edge that connects two basic blocks. For example, e_2 indicates that the decision in s_1 has been evaluated to be *true* in an execution, so it transits from b_1 to b_2 . Since the fault lies in b_2 , we use a weighted border to highlight the rectangular box b_2 . Note



that we add a dummy block $b4$ (as a dashed rectangular box) and an edge $e6$ (as a dashed arrow) to make this CFG more comprehensible.

Statement	Basic block	Control flow graph
<code>if (block_queue) {</code>	$s1$	$b1$
<code> count = block_queue->mem_count + 1;</code>	$s2$	Control Flow Graph (CFG) for the code excerpt on the left: 
<code> n = (int) (count*ratio);</code>	$s3$	
<code> proc = find_nth(block_queue, n);</code>	$s4$	
<code> if (proc) {</code>	$s5$	
<code> block_queue = del_ele(block_queue, proc);</code>	$s6$	
<code> prio = proc->priority;</code>	$s7$	$b3$ (block whose execution leads to failure)
<code> prio_queue[prio] =</code>	$s8$	(We add a dummy block $b4$ containing statement $s9$, and an edge $e6$ to make a complete CFG.)
<code> append_ele(prio_queue[prio], proc);</code>	$s9$	
<code> }</code>		
<code> // next basic block</code>		

susp.: suspiciousness of a statement/edge in relation to a fault

rank: ranking of a statement/edge in the generated list

Basic block	Test case							Tarantula		SBI		Jaccard		Branch		CP	
	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	susp.	rank	susp.	rank	susp.	rank	susp.	rank	susp.	rank
$b1$	•	•	•	•	•	•	•	0.50	9	0.29	9	0.29	9	0.71	9	0.11	9
$b2$ (fault)	•	•	•	•	•	•	•	0.71	7	0.50	7	0.50	7	0.71	9	1.11	4
$b3$ (block whose execution leads to failure)	•	•	•	•	•	•	•	0.71	7	0.50	7	0.50	7	0.71	9	1.11	4
	•	•	•	•	•	•	•	0.71	7	0.50	7	0.50	7	0.71	9	1.11	4
	•	•	•	•	•	•	•	0.71	7	0.50	7	0.50	7	0.71	9	1.11	4
$b4$	•	•	•	•	•	•	•	0.50	9	0.29	9	0.29	9	0.71	9	0.11	9
Edge																	
$e1$	•	•	•	•	•	•	•							0.53		0.00	
$e2$	•	•	•	•	•	•	•							0.71		0.43	
$e3$	•	•	•	•	•	•	•							0.00		-1.00	
$e4$	•	•	•	•	•	•	•							0.71		1.00	
$e5$	•	•	•	•	•	•	•							0.41		0.11	
$e6$	•	•	•	•	•	•	•							N/A		1.00	
Pass/fail	P	P	F	P	P	F	P										
% of code examined according to the ranking of statements								78%	78%	78%	78%	100%	44%				
Ranking order of basic blocks								$b3$ before $b2$		$b2$ and $b3$ have the same rank				$b2$ before $b3$			

Figure 3.1 Motivating example

We examine the program logic, and observe that many failures are caused by the execution of $b2$ followed by $b3$, rather than merely executing $b2$ without executing $b3$. Even if $b2$ is executed and results in some infected program state, skipping $b3$ will not alter the priority queue, and thus the effect of the fault at $s2$ is less likely to be observed through subsequent program execution. On the other hand, executing $b2$ followed by $b3$ means that an infected program state (such as incorrect values for the variables $count$, n , or $proc$) at $b2$ is successfully propagated to $b3$ through the edge $e4$. Since previous studies suggest the comparison of execution statistics to assess the suspiciousness scores of statements, they will be more likely to result in a wrong decision — $b3$ will appear to be more suspicious than $b2$. The following serves as an illustration.

In this example, we use seven test cases (dubbed $t1$ to $t7$). Their statement and edge execution details are shown in Figure 3.1. A cell with the “●” notation indicates that the corresponding statement is exercised (or an edge is traversed) in the corresponding test execution. For instance, let us take the first test case $t1$ (a successful one, referred to as “passed”). During its execution, the basic blocks $b1$, $b2$, and $b4$ are exercised; moreover, the control flow edges $e1$, $e2$, and $e5$ are traversed. Other test cases can be interpreted similarly. The passed/fail status of each test case is shown in the “Pass/Fail status” row. We apply *Tarantula* [68], *Jaccard* [1], and *SBI*¹ [112] to compute the suspiciousness score of every statement, and rank statements in descending order of their scores. Presuming that the programmer may check each statement according to their ranks until reaching the fault [68][112], we thus compute the effort of code examination to locate this fault [68][112]. We show their effectiveness in the “susp.” and “rank” columns, and the row “% of code examined according to the ranking of statements” of Figure 3.1. We observe that $b3$, rather than $b2$, is deemed to be the most suspicious basic block if we apply *Tarantula* or *SBI*. When applying *Jaccard*, $b2$ and $b3$ are equally deemed to be the most suspicious basic blocks. As a result, the fault *cannot* be effectively located by *any* of these techniques. To locate the fault, each examined technique needs to examine 78% of the code.

Intuitively, the execution of $b3$ may lead to a failure, and yet it is not the fault. On the other hand, $b2$ contains the fault, but its execution does not lead to a failure as often as $b3$. Since existing techniques find the suspicious program entities that correlate to failures, they give higher ranks to those statements (such as $b3$) whose executions frequently lead to failures, but give lower ranks to those statements (such as $b2$) whose executions less often lead

¹ In this thesis, we use the term *SBI* to denote Yu et al.’s approach [112] of applying Liblit et al.’s work *CBI* [75] at statement level. At the same time, we still keep the term *CBI* when referring to the original technique in [75].



to failures. If we always separately assess the fault suspiciousness of individual statements (such as $b2$ and $b3$) and ignore their relations, this problem may hardly be solved.

Since executing an edge can be regarded as executing both the two basic blocks connected by the edge, do edge-oriented techniques somehow capture the relationships among statements and perform effectively on this example? We follow [96] to adopt *br*, an edge-oriented technique (which we will refer to as *Branch* in this thesis) to work on this example. *Branch* assesses the suspiciousness scores of control flow edges (say, $e1$ and $e2$), and then associate their suspiciousness scores with statements that are directly connected (in sense of incoming edges or outgoing edges). *Branch* first ranks $e2$ and $e4$ as the most suspicious edges (both having a suspiciousness score of 0.71). In a program execution, traversing $e2$ means to enter the true branch of $s1$ followed by executing $b2$, and traversing $e4$ means having executed $b2$ and will continue to execute $b3$. We observe that executing $b2$ generates infected program states (on variables *count*, *n*, and *proc*), which propagate to $b3$ through $e4$. We further observe that these two highly ranked edges precisely pinpoint the fault location. When associating edges to statements, the rules in *Branch* only propagate the edge suspiciousness to those statements within the same block as the conditional statement for the edge. However, a fault may be several blocks away from the edge and the loop construct may even feedback a faulty program state. For this example, *Branch* assigns identical suspiciousness scores to all statements and they cannot be distinguished from one another. As a result, 100% code examination effort is needed to locate the fault. Since the core of *Branch* is built on top of *Ochiai* [1], we have iteratively replaced this core part of *Branch* by *Tarantula*, *Jaccard*, and *SBI*. However, the fault-localization effectiveness results are still unsatisfactory (100% code to be examined). In practice, the propagation of infected program states may take a long way, such as a sequence of edges, before it may finally result in failures. We need a means to transfer over the edges information about infected program states and failures.

3.3 Our Fault-localization Model

3.3.1 Problem Settings

Let P be a faulty program, $T = \{t_1, t_2, \dots, t_u\}$ be a set of test cases associated with passed executions, and $T' = \{t'_1, t'_2, \dots, t'_v\}$ be a set of test cases that are associated with failed executions. In the motivating example, for instance, P is version v2 of schedule, $T = \{t1, t2, t4, t5, t7\}$, and $T' = \{t3, t6\}$.



Similar to existing work (such as [68]), the technique assesses the fault suspiciousness of each statement of P and can also produce a list of statements of P in descending order of the suspiciousness scores.

3.3.2 Preliminaries

We use $G(P) = \langle B, E \rangle$ to denote the control flow graph (CFG) [5][7] of the program P , where $B = \{b_1, b_2, \dots, b_n\}$ is the set of basic blocks (*blocks* for short) of P , and $E = \{e_1, e_2, \dots, e_m\}$ is the set of control flow edges (*edges* for short) of P , in which each edge e_i goes from one block to another (possibly the same block) in B . Thus, we sometimes write e_i as $\text{edg}(b_{i1}, b_{i2})$ to denote an edge going from b_{i1} to b_{i2} ; this edge e_i is called the *incoming* edge of b_{i2} and the *outgoing* edge of b_{i1} . The block b_{i2} is a *successor block* of b_{i1} , and the block b_{i1} is a *predecessor block* of b_{i2} . We further use the notation $\text{edg}(*, b_j)$ and $\text{edg}(b_j, *)$ to represent the set of incoming edges and the set of outgoing edges of b_j , respectively. In Figure 3.1, for instance, edges e_4 and e_5 are the outgoing edges of block b_2 , and block b_3 is a successor block of b_2 with respect to edge e_4 ; $\text{edg}(b_3, *) = \{e_6\}$ is the set of outgoing edges of block b_3 , and $\text{edg}(*, b_3) = \{e_4\}$ is the set of incoming edges of block b_3 .

An edge is said to have been *covered* by a program execution if it is traversed at least once. For every program execution of P , whether an edge is covered in E can be represented by an edge profile. Suppose t_k is a test case. We denote the edge profile for t_k by $P(t_k) = \langle \theta(e_1, t_k), \theta(e_2, t_k), \dots, \theta(e_m, t_k) \rangle$, in which $\theta(e_i, t_k)$ means whether the edge e_i is covered in the corresponding program execution of t_k . In particular, $\theta(e_i, t_k) = 1$ if e_i is covered by the execution, whereas $\theta(e_i, t_k) = 0$ if e_i is not covered. Take the motivating example in Section 3.2 for illustration. The edge profile for test case t_1 is $P(t_1) = \langle \theta(e_1, t_1), \theta(e_2, t_1), \theta(e_3, t_1), \theta(e_4, t_1), \theta(e_5, t_1), \theta(e_6, t_1) \rangle = \langle 1, 1, 0, 0, 1, 0 \rangle$.

3.3.3 Our Model -- CP

We introduce our fault-localization model in this section.

A program execution passing through an edge indicates that the related program states have propagated via the edge. Therefore, we abstractly model a program state in a program execution as whether the edge has been covered by the program execution, and contrast the edge profiles in passed executions to those of failed executions to **C**apture the suspicious **P**ropagation of program states abstractly. We name our model as **CP**.



CP first uses equation (3.1) to calculate the *mean edge profile* for the corresponding edge profiles of all passed executions, and another mean edge profile for those of all failed executions. Such mean edge profiles represent the central tendencies of the program states in the passed executions and failed executions, respectively. *CP* contrasts the two mean edge profiles to assess the suspiciousness of every edge. The formula to compute the suspiciousness score is given in equation (3.2) and explained with the aid of equation (3.3).

During a program execution, a block may propagate program states to adjacent blocks via edges connecting to that block. We then use equation (3.4) to calculate the ratio of the propagation via each edge, and use such a ratio to determine the fraction of the suspiciousness score of a block propagating to a predecessor block via that edge. We use backward propagation to align with the idea of backtracking from a failure to the root cause. For each block, *CP* uses a linear algebraic equation to express its suspiciousness score by summing up such fractions of suspiciousness scores of successor blocks of the given block. Such an equation is constructed using equation (3.5) or (3.6), depending on whether the block is a normal or exit block. By solving the set of equations (by Gaussian elimination), we obtain the suspiciousness score for each block involved.

As presented later, *CP* ranks all blocks in descending order of their suspiciousness scores, then assigns a rank to each statement, and produces a ranked list of statements by sorting them in descending order of their suspiciousness scores.

Calculating the Edge Suspiciousness Score

In Section 3.2, we have shown that edges can provide useful correlation information for failures. However, the size of T may be very different from that of T' . To compare the sets of edge profiles for T with those for T' , we propose to compare their arithmetic means (that is, central tendencies).

If an edge is never traversed in any execution, it is irrelevant to the observed failures. There is no need to compute the propagation of suspicious program states through that edge. We thus exclude them from our calculation model in Sections 3.3.1 and 3.3.2.

In our model, we use the notation $P^\vee = \langle \theta^\vee(e_1), \theta^\vee(e_2), \dots, \theta^\vee(e_m) \rangle$ to denote the mean edge profile for T , and $P^\times = \langle \theta^\times(e_1), \theta^\times(e_2), \dots, \theta^\times(e_m) \rangle$ for T' , where $\theta^\vee(e_i)$ and $\theta^\times(e_i)$ for $i = 1$ to m are calculated by:

$$\theta^\vee(e_i) = \frac{1}{u} \sum_{t_k \in T} [\theta(e_i, t_k)]; \quad \theta^\times(e_i) = \frac{1}{v} \sum_{t'_k \in T'} [\theta(e_i, t'_k)]. \quad (3.1)$$



Note that the variables u and v in equation (3.1) represent the total numbers of passed and failure-causing test cases, respectively. Intuitively, $\theta^\vee(e_i)$ and $\theta^\times(e_i)$ stand for *the probabilities of an edge being exercised in a passed execution and failed execution, respectively, over the given test set*. For example, $\theta^\times(e_4) = (\theta(e_4, t_3) + \theta(e_4, t_6)) / 2 = (1 + 0) / 2 = 0.5$ and, similarly, $\theta^\vee(e_4) = 0$.

Edge Suspiciousness Calculation

We calculate the suspiciousness score of any given edge e_i using the equation

$$\theta^\Delta(e_i) = \frac{\theta^\times(e_i) - \theta^\vee(e_i)}{\theta^\times(e_i) + \theta^\vee(e_i)}, \quad (3.2)$$

which contrasts the difference between the two mean edge profiles. Intuitively, $\theta^\Delta(e_i)$ models the (normalized) difference between the probability of e_i being traversed in an average passed execution and the probability of an average failed execution. When $\theta^\Delta(e_i)$ is positive, it reflects that the probability of edge e_i being covered in P^\times is larger than that in P^\vee . Since such an edge is more frequently exercised in failed executions than in passed executions, it may be more likely to be related to a fault. When $\theta^\Delta(e_i) = 0$, the edge e_i has identical probabilities of being covered in P^\times and P^\vee . Such an edge is deemed to be less likely to be related to a fault than an edge having a positive suspiciousness score. When $\theta^\Delta(e_i)$ is negative, it means that e_i is less frequently executed in P^\times than in P^\vee .

In short, for an edge e_i , the higher the values of $\theta^\Delta(e_i)$, the more suspicious the edge e_i is deemed to be, and the more suspicious the propagation of program states via e_i is deemed to be.

To understand why equation (3.2) is useful for ranking edges according to their suspiciousness, let $Prob(e_i)$ denote the (unknown) probability that the propagation of infected program states via e_i will cause a failure. The best estimation for this probability is given as

$$Prob(e_i) = \frac{v \cdot \theta^\times(e_i)}{v \cdot \theta^\times(e_i) + u \cdot \theta^\vee(e_i)}. \quad (3.3)$$

The proof is given as follows.

Proof 3.1:



Let $Prob(e_i)$ be the probability that the propagation of infected program states via e_i causes a failure.

Let $T = \{t_1, t_2, \dots, t_u\}$ be a set of test cases associated with passed executions, and $T' = \{t'_1, t'_2, \dots, t'_v\}$ be a set of test cases associated with failed executions. Let $\theta(e_i, t_k)$ denote whether the edge e_i is covered in the corresponding program execution of t_k . We would like to estimate the value of $Prob(e_i)$ from the subsets $T_1 = \{t_k \mid \theta(e_i, t_k) = 1\} \subset T$ and $T_2 = \{t'_k \mid \theta(e_i, t'_k) = 1\} \subset T'$ because the executions in these two subsets correlates with the traversal of e_i . The expected number of failed executions in the sample set of $T_1 \cup T_2$ is $Prob(e_i) \times |T_1 \cup T_2|$. This estimate is unbiased.

To maximize the value of $Prob(e_i)$, we set the expected number of failed executions in the sample set to be equal to the actual number of failed executions. That is, we set $Prob(e_i) \times |T_1 \cup T_2| = |T_1|$. We then solve for $Prob(e_i)$ and obtain equation (3.3). The details of the proof are straightforward. ■

Moreover, no matter whether equation (3.2) or (3.3) is chosen to estimate the fault relevance of edges, sorting edges in descending order of the results always generates the same edge sequence (except tie cases). In other words, we can also determine the order of the suspiciousness of the edges through equation (3.3), or determine the probability that an edge causes a failure by using equation (3.2). The proof is given as follows.

Proof 3.2:

We need an auxiliary function for the proof. We define a sign function such that $\mathbf{sgn}[x] = -1$ if $x < 0$, $\mathbf{sgn}[x] = 0$ if $x = 0$, and $\mathbf{sgn}[x] = 1$ if $x > 0$.

Suppose e_i and e_j are two edges in E satisfying the conditions (i) $\theta^\times(e_i) \neq 0 \vee \theta^\vee(e_i) \neq 0$ and (ii) $\theta^\times(e_j) \neq 0 \vee \theta^\vee(e_j) \neq 0$. We make use of the sign function to express their relative ranking order with respect to equation (3.3) as $\mathbf{sgn}[Prob(e_i) - Prob(e_j)]$. Similarly, we express their relative ranking order with respect to equation (3.2) as $\mathbf{sgn}[\theta^\Delta(e_i) - \theta^\Delta(e_j)]$.

Case 1 ($\theta^\times(e_i) = 0$). By equation (3.3), $Prob(e_i) = 0$. Also by equation (3.3), $Prob(e_j) = \frac{v\theta^\times(e_j)}{v\theta^\times(e_j) + u\theta^\vee(e_j)} \geq 0$. Hence, $\mathbf{sgn}[Prob(e_i) - Prob(e_j)] = -\mathbf{sgn}[\theta^\times(e_j)]$. Similarly, by equation (3.2), $\mathbf{sgn}[\theta^\Delta(e_i) - \theta^\Delta(e_j)] = -\mathbf{sgn}[\theta^\times(e_j)]$. Thus, $\mathbf{sgn}[Prob(e_i) - Prob(e_j)] = \mathbf{sgn}[\theta^\Delta(e_i) - \theta^\Delta(e_j)]$.

Case 2 ($\theta^\vee(e_i) = 0$). Similarly to the proof of case 1, we have $\mathbf{sgn}[Prob(e_i) - Prob(e_j)] = \mathbf{sgn}[\theta^\Delta(e_i) - \theta^\Delta(e_j)]$.



Case 3 ($\theta^\times(e_i) \neq 0 \wedge \theta^\vee(e_i) \neq 0 \wedge (\theta^\times(e_j) = 0$ or $\theta^\vee(e_j) = 0)$). Similarly to case (1), $\mathbf{sgn}[Prob(e_i) - Prob(e_j)] = \mathbf{sgn}[\theta^\Delta(e_i) - \theta^\Delta(e_j)]$.

Case 4 ($\theta^\times(e_i) \neq 0 \wedge \theta^\vee(e_i) \neq 0 \wedge \theta^\times(e_j) \neq 0 \wedge \theta^\vee(e_j) \neq 0$). We first discuss the value of $\mathbf{sgn}\left[\frac{\theta^\times(e_i)}{\theta^\vee(e_i)} - \frac{\theta^\times(e_j)}{\theta^\vee(e_j)}\right]$. Since none of $\theta^\times(e_i)$, $\theta^\vee(e_i)$, $\theta^\times(e_j)$, and $\theta^\vee(e_j)$ is 0, each of them should be a positive number. Suppose a , b , and c are any positive numbers, and d is any number. We have

$$\begin{aligned} \mathbf{sgn}\left[\frac{\theta^\times(e_i)}{\theta^\vee(e_i)} - \frac{\theta^\times(e_j)}{\theta^\vee(e_j)}\right] &= -\mathbf{sgn}\left[\frac{\theta^\vee(e_i)}{\theta^\times(e_i)} - \frac{\theta^\vee(e_j)}{\theta^\times(e_j)}\right] = -\mathbf{sgn}\left[b\frac{\theta^\vee(e_i)}{\theta^\times(e_i)} + a - \right. \\ &b\frac{\theta^\vee(e_j)}{\theta^\times(e_j)} - a\left.] = -\mathbf{sgn}\left[\frac{a\theta^\times(e_i)+b\theta^\vee(e_i)}{\theta^\times(e_i)} - \frac{a\theta^\times(e_j)+b\theta^\vee(e_j)}{\theta^\times(e_j)}\right] = \\ \mathbf{sgn}\left[\frac{\theta^\times(e_i)}{a\theta^\times(e_i)+b\theta^\vee(e_i)} - \frac{\theta^\times(e_j)}{a\theta^\times(e_j)+b\theta^\vee(e_j)}\right] &= \mathbf{sgn}\left[c\frac{\theta^\times(e_i)}{a\theta^\times(e_i)+b\theta^\vee(e_i)} - d - \right. \\ &c\frac{\theta^\times(e_j)}{a\theta^\times(e_j)+b\theta^\vee(e_j)} + d\left.] = \\ \mathbf{sgn}\left[\frac{c\theta^\times(e_i)-d(a\theta^\times(e_i)+b\theta^\vee(e_i))}{a\theta^\times(e_i)+b\theta^\vee(e_i)} - \frac{c\theta^\times(e_j)-d(a\theta^\times(e_j)+b\theta^\vee(e_j))}{a\theta^\times(e_j)+b\theta^\vee(e_j)}\right]. \end{aligned}$$

By setting $a = u$, $b = v$, $c = u$, and $d = 0$, we have $\mathbf{sgn}\left[\frac{\theta^\times(e_i)}{\theta^\vee(e_i)} - \frac{\theta^\times(e_j)}{\theta^\vee(e_j)}\right] = \mathbf{sgn}[Prob(e_i) - Prob(e_j)]$. Similarly, by setting $a = 1$, $b = 1$, $c = 2$, and $d = 1$, we have $\mathbf{sgn}\left[\frac{\theta^\times(e_i)}{\theta^\vee(e_i)} - \frac{\theta^\times(e_j)}{\theta^\vee(e_j)}\right] = \mathbf{sgn}[\theta^\Delta(e_i) - \theta^\Delta(e_j)]$. Hence, we obtain $\mathbf{sgn}[Prob(e_i) - Prob(e_j)] = \mathbf{sgn}[\theta^\Delta(e_i) - \theta^\Delta(e_j)]$.

Hence, the relative ranking order of any two edges computed by equation (3.2) is the same as that computed by equation (3.3). ■

We will adopt equation (3.2) rather than equation (3.3) because the value range of equation (3.2), which is from -1 to 1 and symmetric with respect to 0 , favors follow-up computation. For example, equation (3.3) always generates positive values. However, summing up positive operands always generates an operation result that is greater than each operand, and hence there may not be a solution for the constructed equation set.

On the other hand, by adopting equation (3.2), the calculated ‘‘probability’’ may have positive, negative, or zero values. Here, the absolute value of such a calculated ‘‘probability’’ value has no intuitive physical meaning of probability. In fact, we only reference these values to determine the relative order of two statements being related to faults.



Take the motivating example as an illustration. $Prob(e1) = (2 \times 1.00) / (2 \times 1.00 + 5 \times 1.00) = 0.29$. Similarly, $Prob(e2) = 0.50$, $Prob(e3) = 0.00$, $Prob(e4) = 1.00$, $Prob(e5) = 0.33$, and $Prob(e6) = 1.00$. Furthermore, $\theta^\Delta(e1) = (1.00 - 1.00) / (1.00 + 1.00) = 0.00$. Similarly, $\theta^\Delta(e2) = 0.43$, $\theta^\Delta(e3) = -1.00$, $\theta^\Delta(e4) = 1.00$, $\theta^\Delta(e5) = 0.11$, and $\theta^\Delta(e6) = 1.00$. By sorting $e1$ to $e6$ in descending order of their values using equation (3.2), we obtain $\langle \{e4, e6\}, e2, e5, e1, e3 \rangle$, where $e4$ and $e6$ form a tie case. Based on the two most suspicious edges ($e4$ and $e6$), we can focus our attention to trace from $b2$ to $b3$ via $e4$, and then to $b4$ via $e6$. However, one still does not know how to rank the fault suspiciousness of the blocks (other than examining all three blocks at the same time). In the next section, we will show how we use the concept of propagation to determine the suspiciousness score of each block.

Solving Block Suspiciousness Score

By contrasting the mean edge profiles of the passed executions and the failed executions, we have computed the suspiciousness of edges in the last section. In this section, we further associate edges with blocks, and assess fault suspiciousness score of every block. To ease our reference, we use the notation $BR(b_j)$ to represent the suspiciousness score of a block b_j .

Let us first discuss how a program execution transits from one block to another. After executing a block b_j , the execution may transfer control to one of its successor blocks. Suppose b_k is a successor block of b_j . The program states of b_j may be propagated to b_k via the edge $\text{edg}(b_j, b_k)$. Rather than expensively tracking the dynamic program state propagation from b_j to b_k , we approximate the expected number of infected program states of b_j observed at b_k as the fraction of the suspiciousness score of b_j from that of b_k . This strategy aligns with our expectation that we only want to observe failures from control flow information (e.g., edge frequencies) rather than data-flow information (e.g., variable values).

Constructing an Equation Set

To determine the fraction mentioned above, we compute the sum of suspiciousness scores of the incoming edges of b_k , and compute the ratio of propagation via the edge $\text{edg}(b_j, b_k)$ as the ratio of the suspiciousness score of this particular $\text{edg}(b_j, b_k)$ over the total suspiciousness score for all edges. The formula to determine this ratio is as follows. In this equation, if the result of the denominator happens to be zero, we use the result of the numerator as the result of the right part.



$$W(b_j, b_k) = \frac{\theta^\Delta(\text{edg}(b_j, b_k))}{\sum_{\forall \text{edg}(*, b_k)} [\theta^\Delta(\text{edg}(*, b_k))]} \quad (3.4)$$

$W(b_j, b_k)$ models the portion of the contribution of $\text{edg}(b_j, b_k)$ with respect to the total contribution by all incoming edges of b_k . Intuitively, it represents the ratio of the observed suspicious program states at b_k that may be prorogated from b_j .

The fraction of the suspiciousness score that b_k contributes to b_j is, therefore, the product of this ratio and the suspiciousness score of b_k (that is, $BR(b_k) \times W(b_j, b_k)$).

In general, a block b_j may have any number of successor blocks. Hence, we sum up such a fraction from every successor block of b_j to give $BR(b_j)$, the suspiciousness score of b_j , thus:

$$BR(b_j) = \sum_{\forall \text{edg}(b_j, b_k)} [BR(b_k) \cdot W(b_j, b_k)] \quad (3.5)$$

Let us take the motivating example for illustration: b_2 has two outgoing edges connecting to two successor blocks b_3 and b_4 , respectively. Its suspiciousness score $BR(b_2)$ is, therefore, calculated as $BR(b_3) \cdot W(b_2, b_3) + BR(b_4) \cdot W(b_2, b_4)$. The propagation rate $W(b_2, b_4)$ is calculated as $(b_2, b_4) = \theta^\Delta(e_5) / (\theta^\Delta(e_3) + \theta^\Delta(e_5) + \theta^\Delta(e_6)) = 0.11 / (-1.00 + 0.11 + 1.00) = 1$. The propagation rate $W(b_2, b_3)$ is calculated as $\theta^\Delta(e_4) / \theta^\Delta(e_4) = 1.00 / 1.00 = 1$.

Handling Exception Cases

We normally calculate the suspiciousness score of a block via its successor blocks. Let us consider an exception case where a block may have no successor. For a block containing, say, a return, break, or `exit()` function call, or for a block that crashes, the program execution may leave the block, or cease any further branch transitions after executing the block. In our model, if a block is found not to transit the control to other successor blocks in the same CFG (as in the case of a return statement or callback function), we call it an *exit* block. Since exit blocks have no successor block, we do not apply equation (3.5) to calculate its suspiciousness score. Instead, we use the equation



$$BR(b_j) = \sum_{\forall \text{edg}(*, b_j)} [\theta^\Delta(\text{edg}(*, b_j))], \quad (3.6)$$

which sums the suspiciousness scores of all the incoming edges to calculate the suspiciousness score of the exit block. Consider the motivating example again. There is no successor for b_4 in the CFG in Figure 3.1. Hence, $BR(b_4) = \theta^\Delta(e_3) + \theta^\Delta(e_5) + \theta^\Delta(e_6) = -1.00 + 0.11 + 1.00 = 0.11$. (We should add that there are alternative ways to model the suspiciousness score for an exit block, such as using the formulas for block-level statistical fault-localization techniques.)

We have constructed an equation of $BR(b_j)$ for every block b_j (including exit blocks). In other words, we have set up an equation set containing n homogenous equations (one for each block) and n variables as the left-hand side of each equation (one for each suspiciousness score of that block). In such a case, the equation set satisfies a necessary condition of being solvable by existing efficient algorithms for solving equation sets (such as Gaussian elimination [125], which we also adopt in the experiment in Section 3.4). In the motivating example, we can set up an equation set $\{BR(b_4) = 0.11, BR(b_3) = 9 \times BR(b_4), BR(b_2) = BR(b_4) + BR(b_3), BR(b_1) = -9 \times BR(b_4) + BR(b_2)\}$. We can solve it to give $BR(b_4) = 0.11, BR(b_3) = 1.00, BR(b_2) = 1.11, \text{ and } BR(b_1) = 0.11$.

An exception is the case of self-looping block without leaving edge, which may make the equation set inconsistent. Such a case maps to the endless loop in program executions. If an equation set comes with inconsistent equations, we use linear least square approach to find a least square solution [125] for the inconsistent equation set.

Synthesizing a Ranked List of Statements

After obtaining the suspiciousness score for every block, we further group those statements not in any block (such as global assignment statements) into a new block, and give it a lower suspiciousness score than that of any other ranked block. We also merge those blocks having identical suspiciousness scores, and produce a ranked list of blocks in descending order of their suspiciousness scores. All the non-executable statements and statements that are not exercised by any given executions are consolidated into one block, which is appended to the end of the ranked list and given the lowest suspiciousness score. Finally, one may assign ranks for statements. Following previous work [68], the rank of a statement is assigned as the sum of total number of



Table 3.1: Statistics of Subject programs

	Real-life versions	Program description	LOC	No. of single-fault versions	No. of test cases
flex	2.4.7–2.5.4	lexical parser	8571–10124	21	567
grep	2.2–2.4.2	text processor	8053–9089	17	809
gzip	1.1.2–1.3	compressor	4081–5159	55	217
sed	1.18–3.02	text processor	4756–9289	17	370

statements in its belonging block and the total number of statements in the blocks prior to its belonging block. The *CP* column of Figure 3.1 shows the ranks of statements by our method, which only needs 44% code examination effort to locate a fault.

Complexity

The space and time complexity of our technique is analyzed as follows. With the same problem settings (u passed executions, v failed executions, n blocks, and m edges), the space complexity is mainly determined by the space needed to maintain the mean edge profiles for the passed executions and the failed executions, and the suspiciousness scores for edges, which are $O(um)$, $O(vm)$, and $O(m)$, respectively. Therefore the space complexity of our technique is $O(um + vm)$. The time complexity is determined by the time used to solve the set of equations. Since Gaussian elimination is adopted, the time complexity of our technique will be $O(n^3)$.

3.4 Experimental Evaluation

3.4.1 Subject Programs

We use four UNIX programs, namely, **flex**, **grep**, **gzip**, and **sed**, as subject programs. They are real-life medium-sized programs, and have been adopted to evaluate other techniques (as in [64][106][115]). We downloaded the programs (including all versions and associated test suites) from SIR [40] on January 10, 2008. Each subject program has multiple (sequentially labeled) versions. Table 3.1 shows the real-life program versions, numbers of lines of statements (LOC), numbers of applicable faulty versions, and the sizes of the test pools. Take the program **flex** as an example. The real-life versions include **flex-2.4.7** to **flex-2.5.4**, and have 8571 to 10124 lines of statements. 21 single-fault versions are used in the experiment. All these faulty versions share a test



suite that consists of 567 test cases. Following [52], we apply the whole test suite as inputs to individual programs.

3.4.2 Peer Techniques

In our experiment, we select five representative peer techniques to compare with our technique. *Tarantula* [70] and *Jaccard* [1] are two statement-level techniques. They are often chosen as alternatives for comparison in other evaluations of fault-localization techniques. *CBI* [75] and *SOBER* [77] are predicate-level techniques. Since they make use of predicates (such as branch decisions) to locate suspicious program positions, which are related to the edge concept in our technique, we decide to compare these techniques with ours. Note that *CBI* originally proposed to use random sampling to collect predicate statistics to reduce overhead. In our evaluation on *CBI*, we sample all the predicates (as in [77]) via *gcov*. In Yu et al.'s work [112], *CBI* is modified to become a statement-level technique (*SBI* [112]), and we also include *SBI* for comparison with our technique. Note that a tie-breaking strategy is included in *Tarantula* as stated in [112]. *CP* uses no tie-breaking strategy in our experiment.

3.4.3 Experimental Setup

Following the documentation of SIR [40] and previous experiments [52][68][75][76], we exclude any single-fault version whose faults cannot be revealed by any test case. This is because both our technique and the peer techniques used in the experiment [1][75][76][77] require the existence of failed executions. Moreover, we also exclude any single-fault version that fails for more than 20% of the test cases [40][52]. Besides, as Jones et al. have done in [68], we exclude those faulty versions that are not supported by our experimental environment and instrumentation tool (we use the Sun Studio C++ compiler and *gcov* to collect edge profile information on program versions). All the remaining 110 single-fault versions are used in the controlled experiment (see Table 3.1).

3.4.4 Effectiveness Metrics

Each of *Tarantula*, *SBI*, *Jaccard*, and *CP* produces a ranked list of all statements. For every technique, we check all the statements in ascending order of their ranks in the ordered list, until a faulty statement is found. The percentage of statements examined (with respect to all statements) is returned as the effectiveness of that technique. This metric is also used in previous



studies [64][112]. We note also that statements having the same rank are examined as a group.

CBI and *SOBER* generate ranked lists of predicates. To the best of our knowledge, the metric T-score [90] is used to evaluate their effectiveness in previous studies [76][77]. T-score uses a program dependence graph to calculate the distance among statements. Starting from some top elements in a ranked list of predicates, T-score conducts breadth-first search among the statements to locate a fault. The search terminates when it encounters any faulty statement, and the percentage of statements examined (with respect to all statements) is returned as the effectiveness of that technique [90]. Since it is reported in [76] that the “top-5 T-score” strategy gives the highest performance for *CBI* and *SOBER*, we follow suit to choose the top-5 predicates and report the top-5 T-score results as their effectiveness in our experiment.

If a fault is in a non-executable statement (such as the case of a code omission fault), dynamic execution information cannot help locate the fault directly. To reflect the effectiveness of a technique, we follow previous studies (such as [64]) to mark the directly infected statement *or* an adjacent executable statement as the fault position, and apply the above metrics.

3.4.5 Experiment Environment and Issues

The experiment is carried out in a Dell PowerEdge 1950 server (4-core Xeon 5355 2.66GHz processors, 8GB physical memory, and 400GB hard disk) serving a Solaris UNIX with the kernel version Generic_120012-14. Our framework is compiled using Sun C++ 5.8.

When applying our technique, an exceptional case is that the denominator in equation (3.4) may be zero. For every occurrence of a zero denominator in the experiment, the tool automatically replaces it by a small constant. 10^{-10} is chosen as the constant, which is less than any intermediate computing result by many degrees of magnitude. We have varied this constant from 10^{-11} to 10^{-9} and compared the effectiveness results of *CP*, and confirmed that the results are the same.

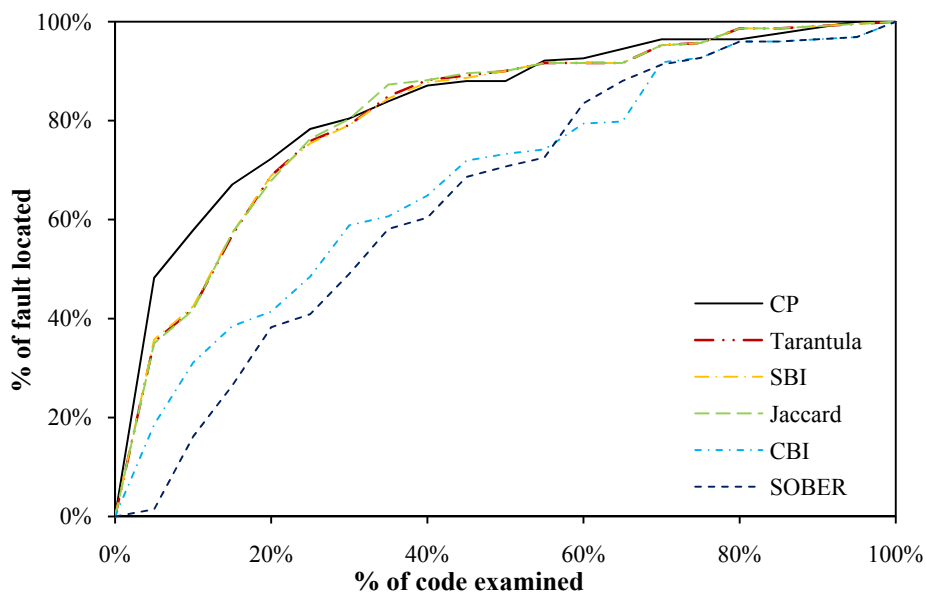
In the experiment, the time needed to generate a ranked list for one faulty version is always less than 1 second. The mean time spent for one faulty version is about 0.455 seconds.

3.4.6 Results and Analysis

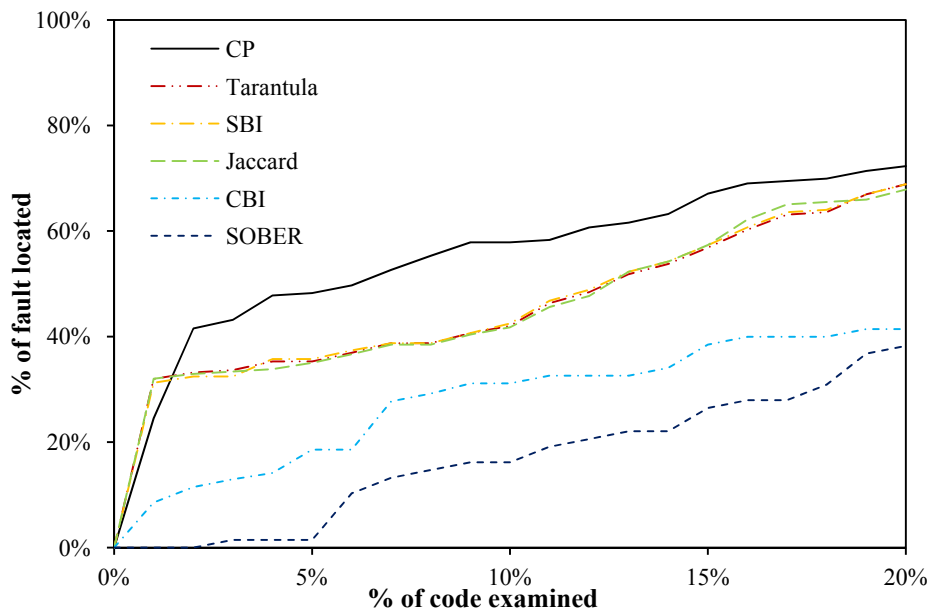
In this section, we compare our technique with *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER*, and report their effectiveness on the 110 single-fault program



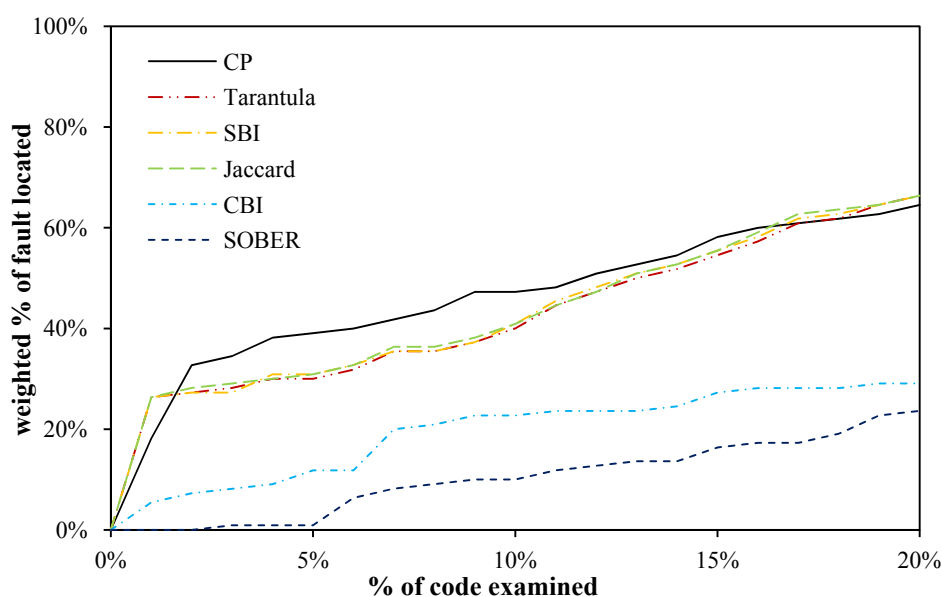
versions. In the following subsections, the data related to “*Tarantula*”, “*SBI*”, “*Jaccard*”, “*CBI*”, and “*SOBER*” are worked out using the techniques described in the papers [112], [112], [35], [1], and [76], respectively. The data related to “*CP*” are worked out using our technique. For every plot in Figure 3.2 and Figure 3.3, we use the same set of x-axis labels and legends.



(a) overall results in full range of [0%, 100%]



(b) overall results in zoom-in range of [0%, 20%]



(c) weighted overall results in range of [0%, 20%]

Figure 3.2: Overall effectiveness comparison

Overall Results

To evaluate the overall effectiveness of a technique, we first take the average of the effectiveness results on the four subject programs. The results are shown in Figure 3.2. In the plot in Figure 3.2(a), the x -axis means the percentage of code that needs to be examined in order to locate the fault (according to the effectiveness metrics). We also refer to it as the *code examination effort* in this thesis. The y -axis means the mean percentage of faults located. Take the curve for *CP* in Figure 3.2(a) for illustration. On average, *CP* can locate 48.24% of all faults by examining up to 5% of the code in each faulty version. The curves of *Tarantula*, *Jaccard*, *CBI*, *SBI*, and *SOBER* can be interpreted similarly. Note that the effectiveness of *Tarantula*, *SBI*, and *Jaccard* are very close, and hence their curves in Figure 3.2 and Figure 3.3 almost completely overlap.

Figure 3.2(a) gives the overall effectiveness of *CP*, *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER*. Each of the six curves starts at the point (0%, 0%) and finally reaches the point (100%, 100%). Obviously, it reflects the fact that no fault can be located when examining 0% of the code, while all the faults can be located when examining 100%. We observe from the figure that *CP* can locate more faults than *CBI* and *SOBER* in the range from 1% to 99% of

the code affordable to be examined. Moreover, the figure also shows that *CP* can locate more faults than *Tarantula*, *SBI*, and *Jaccard* almost in the entire range of the first one third (from 2% to 33%) of the code examination effort.

When comparing the mean effectiveness, although one cannot meaningfully conclude the results from outlier segments (such as those data points beyond three standard deviations), previous studies such as [76] once reported the results on the first 20% code examination range. Therefore, we further zoom in (to the range of [0%, 20%]) as shown in Figure 3.2(b).

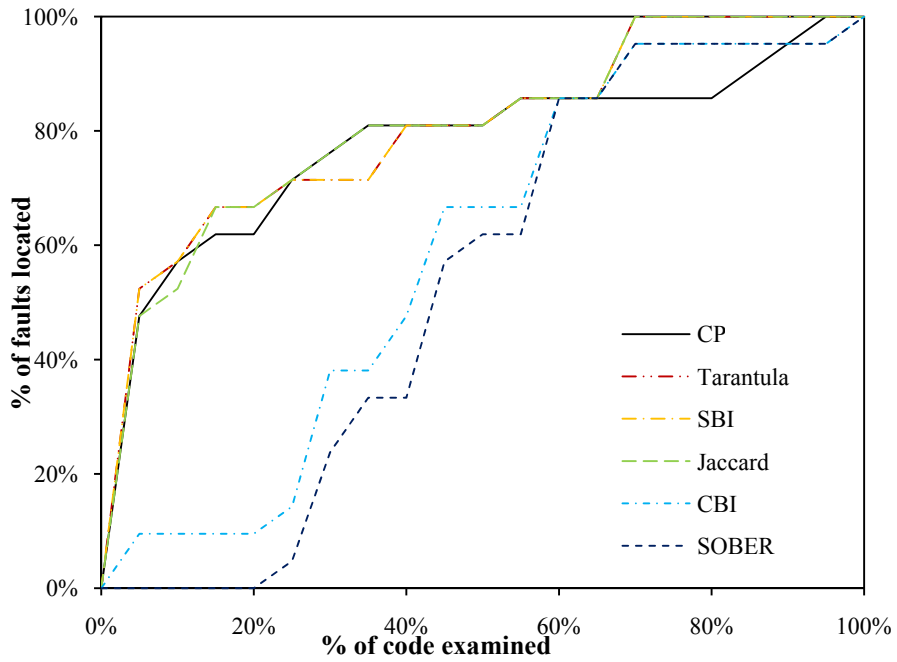
The figure shows that, if only 1% of the code is affordable to be examined, *Tarantula*, *SBI*, and *Jaccard* can locate 31.99% of all faults, *CP* can locate 24.50%, *CBI* can locate 8.54%, while *SOBER* cannot locate any fault. If 2% of the code is affordable to be examined, encouragingly, *CP* not only catches up with *Tarantula*, *SBI*, and *Jaccard*, but also exceeds them a lot. For example, *CP*, *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER* locate 41.55%, 33.18%, 32.45%, 32.90%, 11.48%, and 0.00% of the faults in all faulty versions, respectively. In the remaining range (from 2% to 20%) in Figure 3.2(b), *CP* always locates more faults than the peer techniques. For example, when examining 8%, 9%, and 10% of the code, *CP* locates 55.31%, 57.86%, and 57.86% of the faults, respectively; *Tarantula* locates 38.75%, 40.67%, and 42.03% of the faults; *SBI* locates 38.75%, 40.67%, and 42.49%; and *Jaccard* locates 38.46%, 40.39%, and 41.75%. In summary, by examining up to 20% of the code, *CP* can be more effective than the peer techniques.

In previous studies, a weighted average method has also been used [35]. For example, Chilimbi et al. [35] uses the total number of faults located in all programs as the y -axis (in the sense of Figure 3.2(b)), rather than the average percentage of faults located. To enable reader to compare previously published results with ours, we follow [35] to present such a plot as Figure 3.2(c). From this figure, we observe that if 2% to 16% of the code is examined, *CP* performs better than the other five techniques. However, *Tarantula*, *SBI*, and *Jaccard* catch up with *CP* gradually. The range (21% to 99%) is not shown in this, and yet we do observe that the effectiveness of *CP*, *Tarantula*, *SBI*, and *Jaccard* are very similar. More detailed statistical comparisons can be found in later sections.

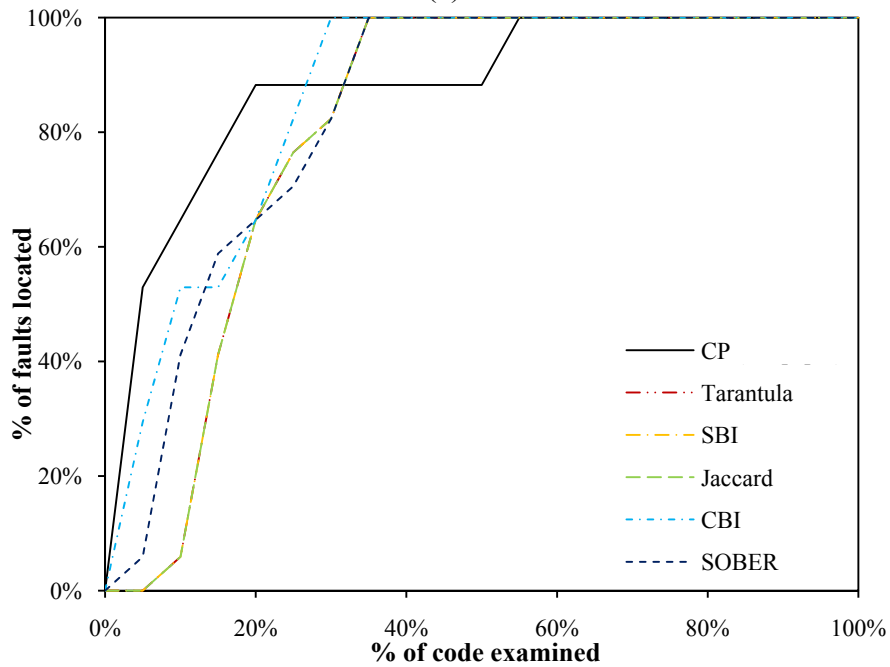
Overall, the experiment shows that *CP* can be effective. At the same time, it also shows that *CP* can be further improved.

Results on Individual Subject Programs



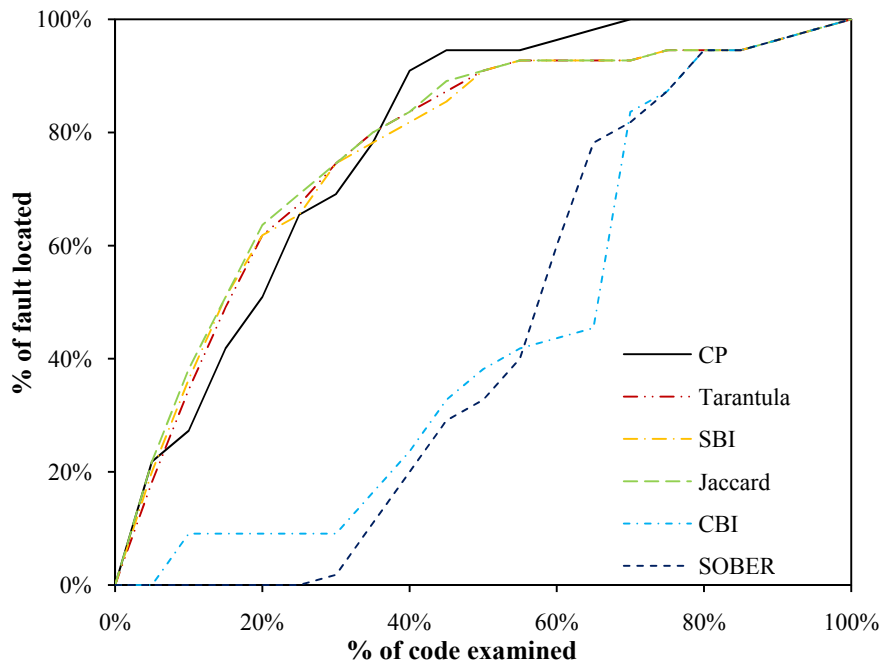


(a) flex

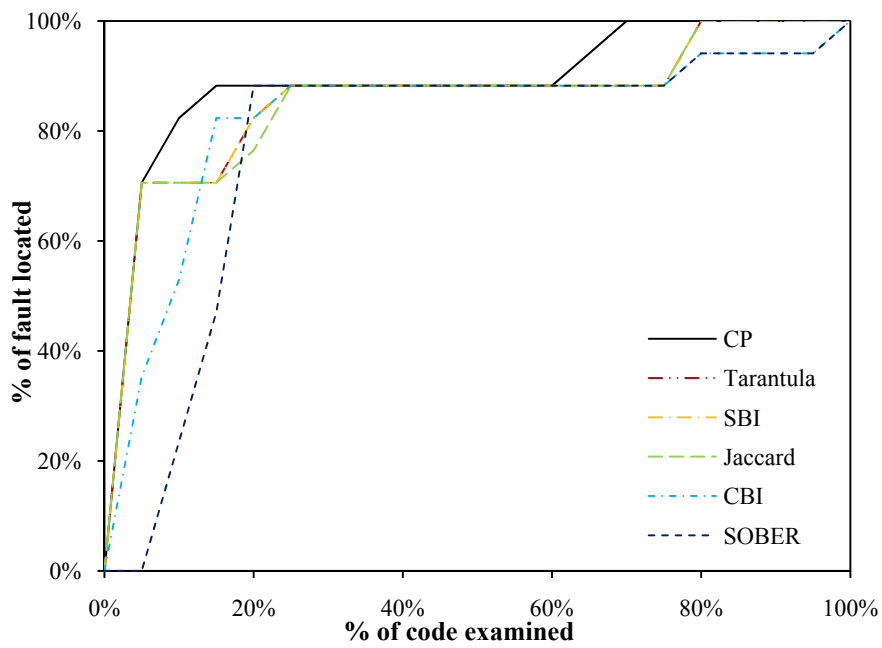


(b) grep





(c) gzip



(d) sed

Figure 3.3: Effectiveness on individual programs.

We further compare the effectiveness of *CP* against the peer techniques on each subject program. Figure 3.3 shows the corresponding results on the programs *flex*, *grep*, *gzip*, and *sed*, respectively. Take the curve for *SBI* in Figure 3.3(a) for illustration. Like Figure 3.2(a), the *x*-axis means the percentage of code examined, and the *y*-axis means the percentage of faults located by *SBI* within the given code examination effort (specified by the respective value on the *x*-axis). The curves for *CP*, *Tarantula*, *Jaccard*, *CBI*, and *SOBER* can be interpreted similarly.

The four plots in Figure 3.3 give the overall effectiveness of *CP*, *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER* on each subject program. If 5% of the code has been examined, *CP* can locate faults in 47.61%, 52.94%, 21.81%, and 70.58% of the faulty versions of the programs *flex*, *grep*, *gzip*, and *sed*, respectively. On the other hand, *Tarantula* can locate 52.38%, 0.00%, 18.18%, and 70.58% of the faults, respectively; *SBI* can locate 52.38%, 0.00%, 20.00%, and 70.58%; *Jaccard* can locate 52.38%, 0.00%, 21.81%, and 70.58%; *CBI* can locate 9.52%, 29.41%, 0.00%, and 35.29%; and *SOBER* can locate 0.00%, 5.88%, 0.00%, and 0.00%. The other points on the curves can be interpreted similarly.

Similarly, let us discuss the first 20% of the code examination range. For *flex* and *gzip*, we observe that *CP* performs better than *CBI* or *SOBER*, and performs comparably with *Tarantula*, *SBI* and *Jaccard*. For *grep* and *sed*, *CP* locates more faults than *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER* within the first 20% code examination range. In summary, *CP* performs outstandingly in this range.

Statistics Analysis on Individual Faulty Versions

In this section, we further use popular statistics metrics to compare different techniques. Table 3.2 lists out the minimum (min), maximum (max), median, mean, and standard derivation (stdev) of the effectiveness of these techniques, on the 110 single-fault versions. The effectiveness of each technique is evaluated using the same metric as in the previous section; therefore, the

Table 3.2: Statistics of effectiveness

	<i>CP</i>	<i>Tarantula</i>	<i>SBI</i>	<i>Jaccard</i>	<i>CBI</i>	<i>SOBER</i>
min	0.01%	0.01%	0.01%	0.01%	0.26%	2.97%
max	93.55%	97.50%	97.50%	97.50%	100.00%	100.00%
median	11.67%	12.86%	12.48%	12.48%	40.74%	42.84%
mean	17.98%	19.63%	19.74%	19.26%	40.55%	42.96%
stdev	20.92%	22.47%	22.63%	22.39%	27.89%	23.98%



Table 3.3: Statistics of differences in effectiveness

	Difference (percentage difference)				
	<i>CP-Tarantula</i>	<i>CP-SBI</i>	<i>CP-Jaccard</i>	<i>CP-CBI</i>	<i>CP-SOBER</i>
< -1%	47 (42.72%)	48 (43.63%)	46 (41.81%)	86 (78.18%)	95 (86.36%)
-1% to 1%	19 (17.27%)	18 (16.36%)	19 (17.27%)	6 (5.45%)	2 (1.81%)
> 1%	44 (40.00%)	44 (40.00%)	45 (40.90%)	18 (16.36%)	13 (11.81%)
< -5%	42 (38.18%)	41 (37.27%)	40 (36.36%)	80 (72.72%)	91 (82.72%)
-5% to 5%	41 (37.27%)	43 (39.09%)	42 (38.18%)	16 (14.54%)	7 (6.36%)
> 5%	27 (24.54%)	26 (23.63%)	28 (25.45%)	14 (12.72%)	12 (10.90%)
< -10%	31 (28.18%)	31 (28.18%)	30 (27.27%)	71 (64.54%)	82 (74.54%)
-10% to 10%	60 (54.54%)	60 (54.54%)	60 (54.54%)	28 (25.45%)	18 (16.36%)
> 10%	19 (17.27%)	19 (17.27%)	20 (18.18%)	11 (10.00%)	10 (9.09%)

smaller the magnitude, the better is the effectiveness. We observe that in each row, *CP* gives the best (smallest) value among the six techniques, which further strengthens our belief that *CP* can be effective on locating faults.

To further find the relative merits on individual versions, we compute the difference in effectiveness between *CP* and each peer technique, and the results are shown in Table 3.3. Take the cell in column “*CP-Tarantula*” and row “< -5%” as an example. It shows that, for 42 (38.18%) of the 110 faulty versions, the code examination effort of using *CP* to locate a fault is less than that of *Tarantula* by more than 5%. Similarly, for the row “> 5%”, only 27 (24.54%) of the 110 versions, the code examination effort of *CP* is greater than that of *Tarantula* by more than 5%. For 41 (37.27%) of the faulty versions, the effectiveness between *CP* and *Tarantula* cannot be distinguished at the 5% significance level.

We therefore deem that, at the 5% significance level, the probability of *CP* performing better than *Tarantula* on these subject programs is higher than that of *Tarantula* performing better than *CP*. We further vary the significance level from 5% to 1% and 10% to produce the complete table. The experimental result shows that the probability of *CP* performing better than its peer technique is consistently higher than that for the other way round.

Discussions of Multi-fault Programs

In this section, we use a real-life multi-fault program to validate the effectiveness of *CP*. Our objective here is to study *CP* rather than comparing *CP* with peer techniques.

Version v3 of flex has the largest number of feasible faults (9 in total) and flex is the largest subject program in the entire experiment. Therefore, we enable all the nine faults of this version to simulate a 9-fault program. Part of the code excerpt is shown in Figure 3.4. After enabling all nine feasible faults,



we execute the test pool in the 9-fault program. It results in 136 failed executions and 431 passed executions.

We apply *CP* to this 9-fault program, and locate the first fault in line 3369 after examining 0.11% of the code. This fault is on an incorrect logical operator. By analyzing the faulty Boolean expression, we find that the fault is enabled only if the decision of the Boolean expression is *true*. As such, this edge (namely, the true decision made in line 3369) rightly reveals failures due to the fault, and *CP* locates this fault effectively. We simulate the fixing of this fault by reverting the statement to the original version. We rerun all the test cases, and find that failed executions have been reduced to 123. We re-apply *CP* and locate the second fault in line 620 after examining 1.21% of the code. The fault is an incorrect assignment of the variable `yy_chk`. In this version, the first statement that uses the variable `yy_chk` is in line 985; it is the root cause of failures. We manually examine the corresponding CFG between the block (dubbed b_a) containing the statement in line 620 and the block (dubbed b_b) containing the statement in line 985. There are many blocks and edges. We observe that, since none of them uses or redefines `yy_chk`, the infected program state of b_a has successfully been propagated to b_b along the edges. Finally, even though the statement that outputs the failure is far away from the fault position, *CP* successfully locates the fault. According to previous studies [47], both of these two faults frequently occur in C programs. *CP* seems to be effective in locating certain popular faults, although more experiments are required to confirm this conjecture.

For space reason, we do not describe the remaining faults in detail. The next six faults are located in lines 1030, 1361, 1549, 3398, 2835, and 11058, respectively. The code examination efforts for locating them are 1.12%, 8.50%, 7.25%, 21.19%, 13.82%, and 88.2%, respectively. The last fault, which results in 6 failures among 567 test cases, is found in line 12193. It is an incorrect

```

620    do_yywrap = ...; // Fault F_AA_4
    ...
985    if ( ! do_yywrap )
    ...
3369  if ( ( need_backing_up && ! nultrans ) ... ) // Fault F_AA_3
    ...
    static yyconst short int yy_chk[2775] = { ...
11825    836, 836, 599, ... // Fault F_AA_2 ...
    }
    ...
12193 while ( yy_chk [...] != ... )

```

Figure 3.4: Excerpts from multi-fault program.



static variable definition. Since this fault is seeded in a global definition statement and the compiler tool `gcov` fails to log its execution, we mark its directly affected statement (say, line 12193) as the fault position. However, *CP* needs to examine 93.55% of the code to locate this fault. We scrutinize the case and find it to be a coincidence. For 7 out of 567 test cases that do not reveal failures, this branch statement is never covered. For the 6 test cases that reveal failures and the remaining 560 passed test cases, both the true branch and the false branch are covered. For more than 90% of the cases, the number of times that each branch is covered is very close to each other (with less than 5% difference). It is hard for *CP* to distinguish these two edges. We view that this practical scenario provides a hint for further improving *CP*, even though the current experiment shows that *CP* is promising.

3.4.7 Threats to Validity

Construct Validity

We used `gcov` to implement our tool in which coverage profiling is completely conducted. The generation of the equation set by the tool is relatively straightforward. The equations are solved using a standard Gaussian elimination implementation. We have implemented the peer techniques ourselves and checked that the implemented algorithms adhere strictly to those published in the literature. We have also conducted trial runs on toy programs with limited test cases to assure the implementations of *CP* and other peer techniques.

Internal Validity

Currently, we follow [76][77] to use T-score when evaluating *CBI* and *SOBER*. Some researchers have reported limitations in T-score (see [36], for example). A P-score has been proposed in [119] and may be considered for future studies.

CP, *Tarantula*, *SBI*, and *Jaccard* produce ranked lists of statements, while *CBI* and *SOBER* generate ranked list of predicates. Consequently, the experiment has used two effectiveness metrics to report the results of different techniques. It is unsuitable to compare *CP* on a par with *CBI* and *SOBER*. In this connection, the comparison and the discussion of *CP* in relation to *CBI* and *SOBER* should be interpreted carefully.



External Validity

We use flex, grep, gzip, and sed as well as their associated test suites to evaluate our technique. These programs are real-life programs with realistic sizes, and they have been used in previous studies [64][106][115]. It is certainly desirable to evaluate *CP* further on other real-life subject programs and scenarios.

3.5 Summary

Existing coverage-based fault-localization approaches use the statistics of test case executions to serve this purpose. They focus on individual program entities, generate a ranked list of their suspiciousness, but ignore the structural relationships among statements. Furthermore, an infected program state triggered by a fault may propagate a long way before it finally causes a failure. Previous techniques, which find program elements whose execution strongly correlates to failure, are not effective to locate this kind of fault.

In this chapter, we have assessed the suspiciousness scores of edges, and set up a set of linear algebraic equations over the suspiciousness scores of basic blocks and statements, which abstractly models the propagation of suspicious program states through control flow edges in a back-tracking manner. Such equation sets is efficiently solved by standard mathematical techniques such as Gaussian elimination and least square method. The empirical results on comparing existing techniques with ours showed that our technique can be effective.

The main contribution of this chapter is twofold. (i) To the best of our knowledge, the work is the first that integrates the propagation of program states to statistical fault-localization techniques. (ii) We conduct the first study that use four real-life medium-sized programs flex, grep, gzip, and sed to sufficiently evaluate five representative techniques, namely Tarantula, SBI, Jaccard, CBI, and SOBER. The empirical results show our method promising.

On the other hand, our work highlights the problem of propagation of infected program states among control flow graph. Intuitively, the long-way propagation of infected program states may increase the difficulty of software debugging. Enhancing the program structure to shorten the potential propagation trace of infected program states may be a suggestion to future programming. However, limited by our knowledge, there is no research thread or related publication in current software debugging researches.



Chapter 4

DES: Statistical Fault-localization Technique at the Level of Evaluation Sequence

In the previous chapter, our experiment studies show that our technique CP is effective. At the same time, we notice that the effectiveness of previous representative techniques (e.g., CBI and SOBER) are not as effective as other statement-level techniques. Therefore, we look into the details of those techniques and make improvements on them.

In this chapter, we first introduce predicate-level fault-localization techniques and our observation of the impact of short-circuit rule on predicate-level fault-localization techniques. After that, we use a motivating example to show how we make use of the evaluation sequence information to perform fault localization at evaluation sequence level. After that, we propose research questions to validate our idea and conduct controlled experiment to answer them.

This chapter is based on our previous work [122], which has been selected for a best paper award in the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008). Part of this chapter is also based on the journal extension [123] of that paper.

4.1 Background

A typical program contains numerous *predicates* in branch statements such as if-and while-statements. (Some programming languages like C further allow predicates on assignment statements.) These predicates are in the form of *Boolean expressions*, such as “`*j <= 1 || src[*i+1] == '\0'`”, which may comprise



further *conditions*, such as “`*j <= 1`” and “`src[*i+1] == '\0'`”. When evaluating a Boolean expression, the “evaluation sequence” structure maintains the execution information of its conditions. In the same example, if the former condition is evaluated as *false*, the latter condition will continue to be evaluated. In the case the latter condition is evaluated as *true*, we use evaluation sequence $\langle false, true \rangle$ to express the sequence of evaluating conditions. On the other hand, if the former condition is evaluated as *true*, the evaluating to the latter condition is short-circuited since solely evaluating the former condition has determined the value of this Boolean expression. In such a case, we use evaluation sequence $\langle true, \perp \rangle$ to express the sequence of evaluating these conditions. Note that the symbol “ \perp ” means the evaluating to a corresponding condition is short-circuited.

Predicate-level statistical fault-localization techniques, however, need to summarize the execution statistics of individual predicates. A compound predicate may be executed in one way or the other owing to short-circuit evaluations over different sub-terms of the predicate. The execution statistics of a compound predicate is, therefore, the summary of a collection of lower-tier evaluations over different sub-terms [122][123]. *Will differentiating such lower-tier evaluations improve the effectiveness of predicate-level fault-localization techniques?*

Since a predicate can be semantically modeled as a Boolean expression, the resultant values of a Boolean expression may be calculated from different evaluation sequences or from the whole predicate as one unit. If we ignore the information on evaluation sequences, we may be masking out useful statistics for effective fault localization.

4.2 Motivation

This section shows a motivating study. It enables readers to have a feeling of how the distribution of evaluation biases [77] at the evaluation sequence level can be used to pinpoint a fault-relevant predicate, which also happens to be a faulty predicate in this example.

The upper part of Figure 4.1 shows a code fragment excerpted from the original version (version v0) of `print_tokens2` from the Siemens suite of programs [40]. We have labeled the three individual conditions as C_1 , C_2 , and C_3 , respectively. The lower part of the same figure shows the code fragment excerpted from a faulty version (version v8) of the Siemens suite, where a fault has been seeded into the predicate by adding an extra condition “`ch == '\t'`”. We have labeled this condition as C_4 .

Because of the effect of short-circuit rules of the C language on Boolean ex-



```

/* Original Version v0 */
if( $\overbrace{ch == ' ' }^{C_1} \parallel \overbrace{ch == '\n' }^{C_2} \parallel \overbrace{ch == 59}^{C_3}$ )
    return(true);

/* Faulty Version v8 */
if( $\overbrace{ch == ' ' }^{C_1} \parallel \overbrace{ch == '\n' }^{C_2} \parallel \overbrace{ch == 59}^{C_3} \parallel \overbrace{ch == '\t' }^{C_4}$ )
    return(true);

```

Figure 4.1: Code excerpts from versions v0 and v8 of print_tokens.

Evaluation sequence	C_1	C_2	C_3	C_4	v0	v8	v0 = v8?
es_1	<i>true</i>	\perp	\perp	\perp	<i>true</i>	<i>true</i>	yes
es_2	<i>false</i>	<i>true</i>	\perp	\perp	<i>true</i>	<i>true</i>	yes
es_3	<i>false</i>	<i>false</i>	<i>true</i>	\perp	<i>true</i>	<i>true</i>	yes
es_4	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>no</i>
es_5	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>		<i>false</i>	yes

Table 4.1: Evaluation sequences of code fragments.

pressions, a condition in a Boolean expression may be evaluated to be *true* or *false*, or may not be evaluated at all (\perp). Furthermore, in terms of evaluations, the conditions on a Boolean expression can be seen as an ordered sequence. In most cases, when a preceding condition in an evaluation sequence is not evaluated, by the short-circuit rule, no succeeding condition in the evaluation sequence will be evaluated.

For the faulty Boolean expression in the fragment shown in Figure 4.1, there are five legitimate evaluation sequences es_1 to es_5 , as shown in Table 4.1. The columns under the individual conditions C_1 to C_4 represent the evaluation outcomes of the respective conditions based on the short-circuit rules of the programming language. In the column entitled v0, it shows the respective resultant values of the predicate in the original version of the program. In this column, the last two grids are merged because the two evaluation sequences (es_4 and es_5) make no difference in the original program. The column entitled v8 shows the respective resultant values in the faulty program. The rightmost column shows whether the original and faulty predicates give the same values.

To gain an idea of whether short-circuit rules can be useful for fault localization, we have run an initial experiment. We apply the whole test pool for



the program from the Software-artifact Infrastructure Repository (SIR) [40], and record the counts of each of the five evaluation sequences for each test case.

Definition 4.2.1 (Evaluation Bias [77]). *Let n_t be the number of times that a predicate P has been evaluated to be true in an execution, and n_f the number of times that it has been evaluated to be false in the same execution. $\pi(P) = \frac{n_t}{n_t+n_f}$ is called **evaluation bias** of predicate P in this particular execution.*

Following [76], we use the formula $\frac{n_t}{n_t+n_f}$ to calculate the evaluation biases (Definition 4.2.1) for the set of passed test cases, and those for the set of failure-causing test cases. The results are shown as the histograms in Figures 4.2 and 4.3. The distribution of evaluation biases over passed test cases and that over failure-causing test cases are shown side by side for comparison. Figures 4.2(a) to 4.2(d) are the comparative distributions of the five evaluation sequences. Figures 4.3(b) and 4.3(c) are the comparative distributions for the whole predicate (when evaluated to be *true* and when evaluated to be *false*, respectively), as used in [76].

From the histograms in Figures 4.2 and 4.3, we observe that the distribution of evaluation biases for es_4 on passed test cases is drastically different from that of the failure-causing ones. Indeed, it is the most different one among all pairs of histograms shown in the figures. We also observe from Table 4.1 that the fault in the code fragment can only be revealed when es_4 is used, and the fault does not affect the values in the other alternatives.

Our initial study indicates that it may be feasible to use evaluation sequences to identify a faulty predicate more accurately. When a fault is not on the predicate, most predicate-based techniques facilitate fault localization by ranking the predicates in order of their relations to the fault. In such a case, our technique also works by finding the fault-relevant predicates rather than the actual faulty statement. However, it is still uncertain how much the use of evaluation sequences will be beneficial to fault localization. We will formulate our research questions in the next section and then investigate them experimentally in Section 4.5.

4.3 Our Fault-localization Model

As we are interested in studying the impact of short-circuit evaluations and evaluation sequences for statistical fault localization, we need a method to incorporate the fine-grained view into a base technique. Intuitively, this will provide execution statistics that may help statistical fault-localization techniques identify the locations of faults more accurately.



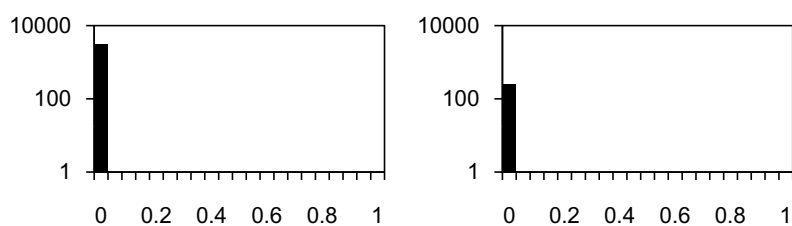
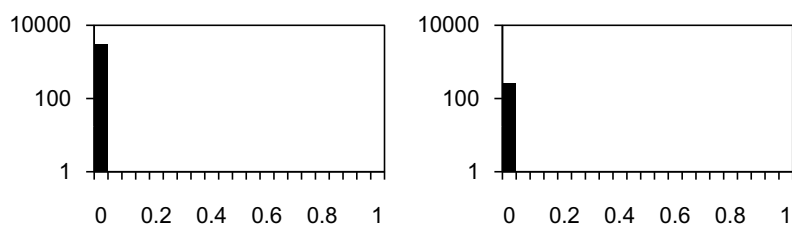
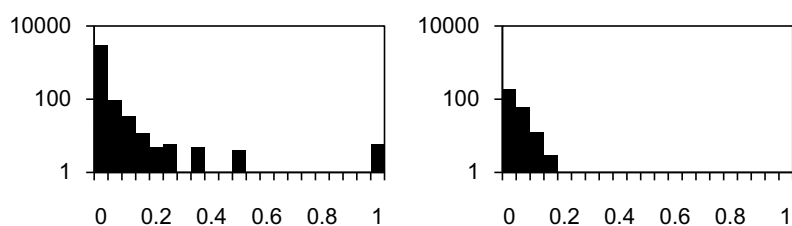
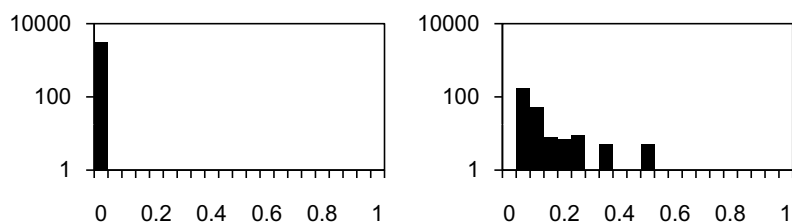
(a) Results of evaluation sequence es_1 (b) Results of evaluation sequence es_2 (c) Results of evaluation sequence es_3 (d) Results of evaluation sequence es_4

Figure 4.2: Comparisons of distributions of evaluation biases for evaluation sequences es_1 , es_2 , es_3 , and es_4 (x -axis: evaluation bias; y -axis: no. of test cases).

We note that a base technique, such as SOBER or CBI, conducts sampling of the predicates in a subject program to collect run-time execution statistics, and ranks the fault relevance of the predicates. To assess the effectiveness of the selected set of predicates to locate faults, researchers may use the t-score [90] metric to determine the percentage of code examined in order to discover the

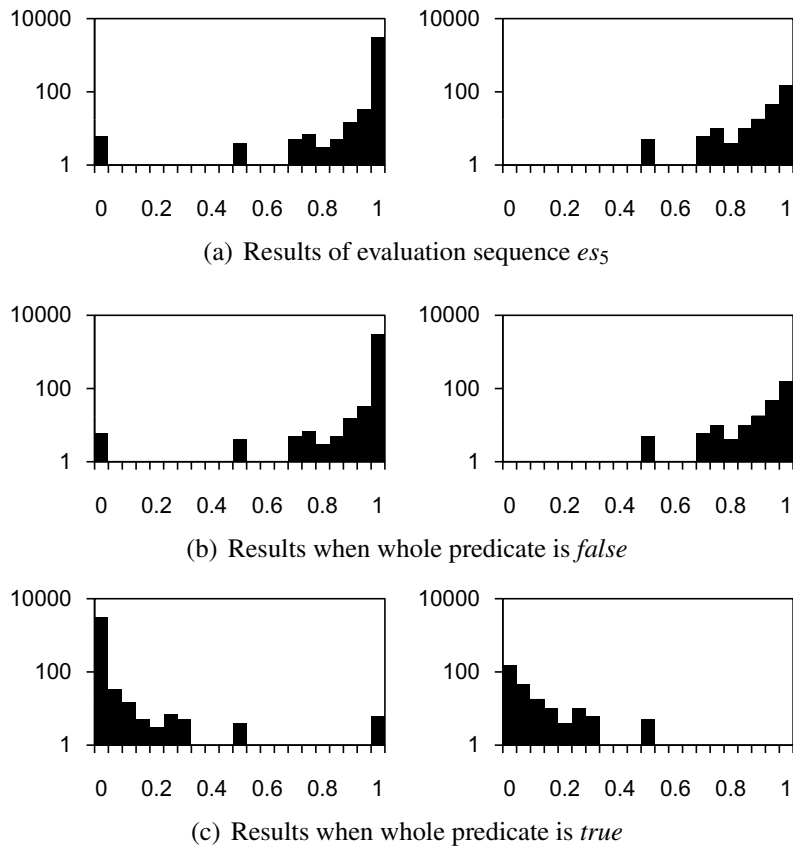


Figure 4.3: Comparisons of distributions of evaluation biases for evaluation sequences es_5 and the whole predicate (x -axis: evaluation bias; y -axis: no. of test cases).

fault. As such, given a set of predicates applicable to a base technique, we identify all the potential evaluation sequences for each of the predicates. We then insert probes at the predicate locations to collect the evaluation outcomes of atomic conditions in these predicates. Based on the evaluation outcomes of the atomic conditions, we can determine the evaluation sequence that takes place for every predicate. For each individual evaluation sequence, we count the number of times it is executed with respect to every test case. By treating each evaluation sequence as a distinct (fine-grained) predicate in the base technique, the ranking approach in the base technique can be adopted to rank these fine-grained predicates.

On the other hand, from the developers' viewpoint, it may be more convenient to recognize (through their eyeballs) the occurrence of an original predicate

from the code, rather than an evaluation sequence of the predicate. Hence, it is to the benefit of developers to map the ranked evaluation sequences to their respective predicates and thus the corresponding statements.

Some measures need to be taken in the above mapping procedure. Different evaluation sequences may receive different ranks. A simple mapping may thus result in a situation where a predicate occurs more than once in a ranking list. We choose to use the highest rank of all evaluation sequences for each individual predicate as the final rank of that predicate. This strategy also aligns with the basic idea of predicate ranking in SOBER and CBI. We refer to the fine-grained approach as *Debugging through Evaluation Sequences (DES)*. Let us take the motivating example in Section 4.2 as an illustration. In previous predicate-based approaches such as SOBER [77], the Boolean expression “`ch == ' ' || ch == '\n' || ch == 59 || ch == '\t'`” is used as one predicate. When the result of the Boolean expression is *true* or *false*, previous techniques evaluate the predicate as *true* or *false*, respectively, and records its evaluation biases accordingly. In our approach, we form a finer-grained viewpoint and investigate four atomic Boolean expressions “`ch == ' '`”, “`ch == '\n'`”, “`ch == 59`”, and “`ch == '\t'`” as shown in Table 4.1. The evaluation sequence es_2 , for instance, shows the case where “`ch == ' '`” is evaluated to be *false*, “`ch == '\n'`” is evaluated to be *true*, and the evaluations of the other two atomic Boolean expressions “`ch == 59`” and “`ch == '\t'`” are short-circuited. Each time the Boolean expression is evaluated, it must fall into one (and only one) of the evaluation sequences es_1 to es_5 . We regard the “falling into” or “not falling into” each evaluation sequence by the Boolean expression as a kind of predicate for that evaluation sequence. For example, if the evaluation of the Boolean expression falls into the evaluation sequence es_2 , we regard the corresponding predicate with respect to es_2 as evaluated to be *true*; and if the evaluation of the Boolean expression falls into another evaluation sequence, we regard the corresponding predicate with respect to es_2 as evaluated to be *false*. We record the evaluation biases for this kind of predicate accordingly, and adapt previous techniques to work on the evaluation sequence level.

4.4 Research Questions

In this section, we will discuss the research questions to be addressed. We refer to a predicate-based statistical fault-localization technique as a *base* technique, and refer to the use of evaluation sequences in predicate execution counts as the *fine-grained* version of the base technique.



- RQ1:** In relation to the base technique, is the use of evaluation sequences for statistical fault localization effective?
- RQ2:** If the answer to **RQ1** is true, is the effectiveness of using evaluation sequences significantly better than the base technique?
- RQ3:** Do the execution statistics of different evaluation sequences of the same predicate differ significantly?

4.5 Experimental Evaluation

This section presents a controlled experiment and its results and analyses.

4.5.1 Subject Programs

In this study, we choose the Siemens suite programs as well as four UNIX utility programs to conduct our experiment.

The Siemens programs were originally created to support research on data-flow and control-flow test adequacy [63]. Our version of the Siemens programs is obtained from the Software-artifact Infrastructure Repository (SIR) [40] at <http://sir.unl.edu>. The Siemens suite consists of seven programs as shown in Table 4.2. A number of faulty versions are attached to each program. In our experiment, if any faulty version comes with no failure-causing cases, we do not include it in the experiment, since the base techniques [75][76] require failure-causing test cases. We use a UNIX tool, `gcov`, to collect the execution statistics needed for computation. Six faulty versions that cannot be processed by `gcov` are excluded. As a result, we use 126 faulty versions in total.

Since the Siemens programs are of small sizes and the faults are seeded manually, we also use medium-sized real-life UNIX utility programs with real and seeded faults as further subjects to strengthen the external validity of our experiment. These programs, also from SIR, include `flex`, `grep`, `gzip`, and `sed` as shown in Table 4.2. Each of these programs has one or more versions and each version contains dozens of single faults. We create one faulty program for each single fault, apply the same strategy above to exclude problematic ones, and use a total of 110 faulty versions as target programs.

Each of the Siemens and UNIX programs is equipped with a test pool. According to the authors' original intention, the test pool simulates a representative subset of the input domain of the program, so that test suites should be drawn



from such a test pool [40]. In the experiment, we follow the work of [76] to input the whole test pool to every technique to rank predicates or their evaluation sequences.

Table 4.2 shows the statistics of the subject programs and test pools that we use. The data with respect to each subject program, including the executable lines of code (column “LOC”), the number of faulty versions (column “# of Versions”), the size of the test pool (column “# of Cases”), the number of Boolean expressions (column “# of Bools”), the average percentage of Boolean expression statements with respect to all statements (column “% of Boo”), and the average percentage of compound Boolean expression statements with respect to all Boolean expression statements (column “% of Com”), are obtained from SIR [40] as at January 10, 2008. Since the subject programs `print.tokens` and `print.tokens2` have similar structures and functionality, and each has only a few faulty versions (which cannot give meaningful statistics), we show their combined results in the figure. (By the same token, the combined results of `schedule` and `schedule2` are shown in Figure 4.7.) For instance, there are 10 faulty versions for the `print.tokens2` program. Their sizes vary from 350 to 354 LoC, and their test pool contains 4115 test cases. On average, 5.4% of the Boolean expression statements in these faulty versions contain compound Boolean expressions. Other rows can be interpreted similarly. We note that many faults in these faulty versions do not occur in predicates.

We observe from the column “% of Com” that, in each subject program, the percentage of predicates having more than one atomic condition is low. This makes the research questions even more interesting: We would like to see whether such a low percentage would affect the performance of a base technique to a large extent.

4.5.2 Experimental Setup

In this section, we describe the setup of the controlled experiment. Using our tool, we produce a set of instrumented versions of the subject programs, including both the original and faulty versions. Based on the instrumentation log as well as the coverage files created by `gcov`, we calculate the execution counts for the evaluation sequences, and finally rank the Boolean expression statements according to the description presented in Section 4.4. We also calculate the number of faults successfully identified through the examined percentage of code at different t-scores (see Section 4.4).

The experiment is carried out on a DELL PowerEdge 1950 server with two 4-core Xeon 5355 (2.66Hz) processors, 8GB physical memory and 400GB hard



Program	LOC	# of Versions	# of Cases	# of Bools	% of Bools	% of Compounds
print_tokens (2 programs)	341–354	17	4130	81	23.7	1.7
replace	508–515	31	5542	66	12.9	2.0
schedule (2 programs)	261–294	14	2710	43	16.4	1.0
tcas	133–137	41	1608	11	8.1	2.4
tot_info	272–274	23	1052	46	16.8	5.6
Average	310	18	3115	55	16.9	3.0
flex (2.4.7–2.5.4)	8571–10124	21	567	969	10.3	5.5
grep (2.2–2.4.2)	8053–9089	17	809	930	10.9	14.1
gzip (1.1.2–1.3)	4081–5159	55	217	591	12.7	11.6
sed (1.18–3.02)	4756–9289	17	370	552	7.8	11.6
Average	7390	28	491	761	10.4	10.7

Legion:

LOC: executable lines of code.

of Versions: no. of faulty versions.

of Cases: no. of test cases in the test pool.

of Bools: average no. of Boolean expressions.

% of Bools: average percentage of Boolean expression statements with respect to all statements.

% of Compounds: average percentage of compound Boolean expressions with respect to all Boolean expressions.

Table 4.2: Statistics of subject programs.

disk equipped, serving Solaris UNIX with the kernel version of Generic_120012-14.

Our experimental platform is constructed using the tools of flex++ 2.5.31, bison++ 1.21.9-1, CC 5.8, bash 3.00.16(1)-release (i386-pc-solaris2.10), and sloc-count 2.26.

4.5.3 Effectiveness Metrics

Effectiveness metrics are widely used to facilitate comparisons among different approaches. Renieris and Reiss [90] propose a metric (t-score) for measuring their fault-localization technique. The method is also adopted by Cleve and



Zeller [36] and Liu et al. [76] to evaluate other fault-localization techniques.

For ease of comparison with previous work, we also use the t-score metric to evaluate the fine-grained evaluation sequence approach in relation to the corresponding base techniques. We select two base techniques for study, namely SOBER [76] and CBI [75], because they are representative predicate-based fault-localization techniques. Both of them take Boolean expressions in conditional statements and loops as predicates and generate a ranked list showing, in descending order, how much each of these predicates is estimated to be related to a fault. Both techniques have been evaluated in a previous study [77] using the t-score metric [90] and compared with other fault-localization techniques. Follow-up studies such as [4] have been derived from these techniques. Since the follow-up work is based on the same framework and hence similar in nature, we will only use the base versions to investigate whether we may improve on them using our approach. If our approach may indeed improve on the two base techniques, their derived versions should also benefit.

In brief, the t-score metric takes a program P , its marked faulty statements S , and a sequence of most suspected faulty statements S' as inputs, and produces a value V as output. The procedure to compute the t-score is as follows: (i) Generate a Program Dependence Graph (PDG) G for P . (ii) Using the dependence relations in the PDG as a measure of distance among statements, do a breadth-first search starting with the statements in S' , until some statement in S is reached. (iii) Return the percentage of searched statements (with respect to the total number of statements in P) as the value V . If the original S' consists of k most suspicious faulty statements, the final result is known as the top- k t-scores.

This measure is useful in assessing objectively the quality of proposed ranking lists of fault-relevant predicates and the performance of fault-localization techniques. Since the evaluation sequence approach is built on top of base techniques (such as SOBER and CBI), we also use t-scores to compare different approaches in our controlled experiment to answer the research questions.

4.5.4 Results and Analysis

In this section, we present the experimental results, compare the relative improvements in effectiveness of the integrated approach with respect to the base techniques, and address the research questions one by one.



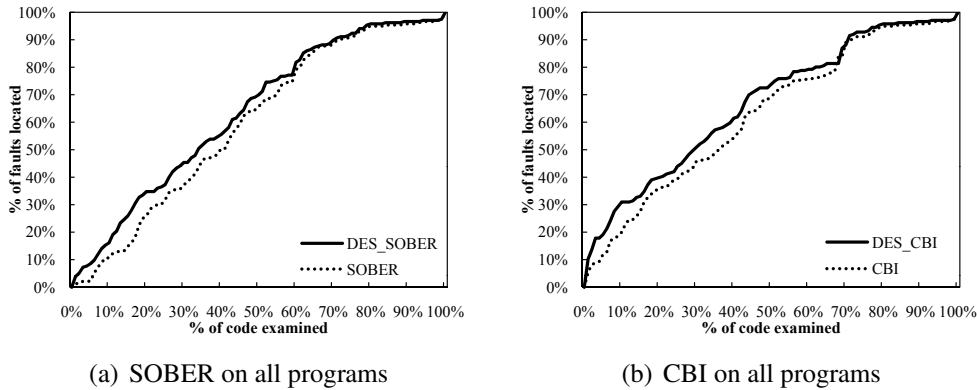


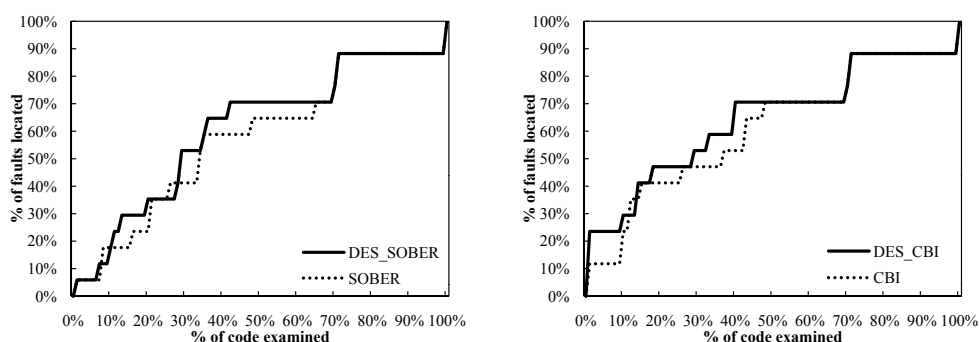
Figure 4.4: Comparisons of DES-enabled techniques with base techniques on all programs.

Overall results of DES-enabled techniques

Figure 4.4(a) compares the results by SOBER and DES-enabled SOBER on all 11 programs, and Figure 4.4(b) compares those by CBI and DES-enabled CBI on the same programs. For ease of discussion, we refer to DES-enabled SOBER as DES_SOBER, and DES-enabled CBI as DES_CBI.

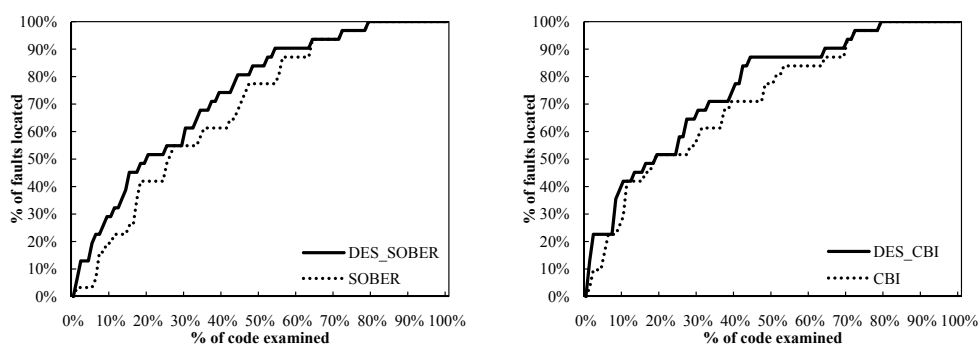
The x -axis of each plot in these two figures shows the t-scores, each of which represent the percentage of statements of the respective faulty program version to be examined. The y -axis is the percentage of faults located within the given code-examination range. According to [76], the use of the top 5 predicates in the ranked list will produce the best results for both SOBER and CBI. For a fair comparison with previous work, we also adopt the use of the top 5 predicates in the controlled experiment. In the remaining parts of this chapter, therefore, we will always compare the top-5 t-scores for DES_SOBER and DES_CBI against those for SOBER and CBI.

We observe from Figure 4.4(a) that DES_SOBER consistently achieves better average fault-localization results (that is, more faults for the same percentage of examined code) than SOBER. For example, when examining 10 percent of the code, DES_SOBER can find about 5% more faults than SOBER. As the percentage of examined code increases, however, the difference shrinks. This is understandable because, when an increasing amount of code has been examined, the difference between marginal increases of located faults will naturally be diminished. When all the faults are located or all the statements are examined, the two curves will attain the same percentage of located faults. We also observe from Figure 4.4(b) that DES_CBI also outperforms CBI. When examining 10



(a) SOBER on print_tokens and print_tokens2 programs (b) CBI on print_tokens and print_tokens2 programs

Figure 4.5: Comparisons of DES-enabled techniques with base techniques on print_tokens and print_tokens2 programs.



(a) SOBER on replace program

(b) CBI on replace program

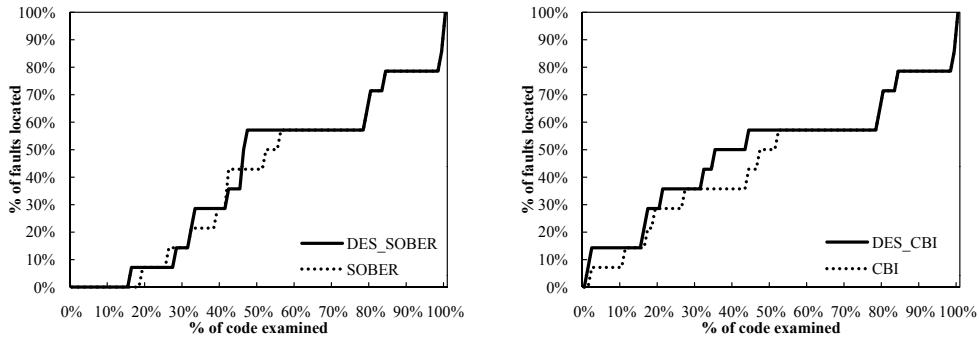
Figure 4.6: Comparisons of DES-enabled techniques with base techniques on replace program.

percent of the code, DES_CBI can find about 10% more faults than CBI.

Individual results of DES-enabled techniques

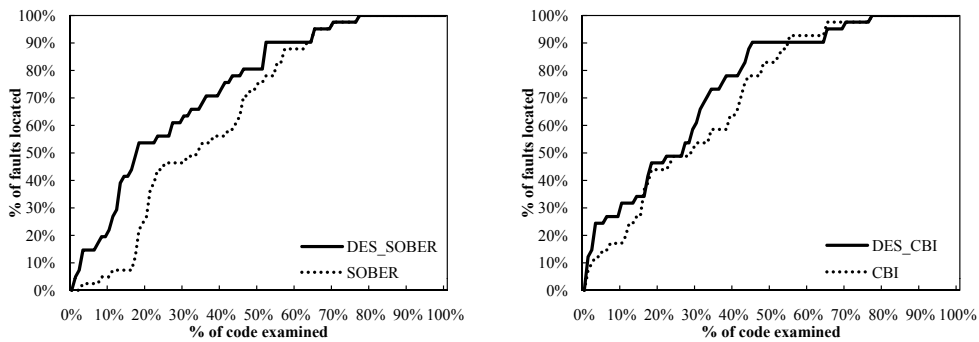
To further verify whether the above results generally hold for all the programs, we examine the outcomes of each individual program, as shown in Figures 4.5 to 4.13.

Let us first focus on Figure 4.5. It shows the results of CBI, DES_CBI, SOBER, and DES_SOBER on the print_tokens and print_tokens2 programs. For



(a) SOBER on schedule and schedule2 programs (b) CBI on schedule and schedule2 programs

Figure 4.7: Comparisons of DES-enabled techniques with base techniques on schedule and schedule2 programs.



(a) SOBER on tcas program (b) CBI on tcas program

Figure 4.8: Comparisons of DES-enabled techniques with base techniques on tcas program.

these two programs, DES_SOBER outperforms SOBER for almost the entire code-examination range from 0 to 100 percent, except two short ranges around 10 percent and 30 percent. Similarly, DES_CBI performs better than CBI almost throughout the range from 0 to 100 percent.

Let us move on to the replace program. DES_SOBER and DES_CBI again exhibit advantage over SOBER and CBI, respectively, almost throughout the entire range from 0 to 100 percent.

For the programs schedule and schedule2, neither DES_SOBER nor SOBER shows advantage over each other. However, for the same programs, DES_CBI



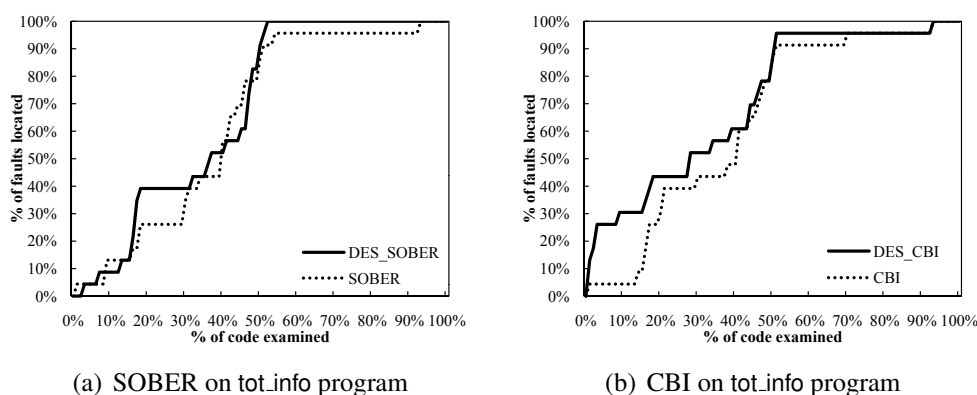


Figure 4.9: Comparisons of DES-enabled techniques with base techniques on tot_info program.

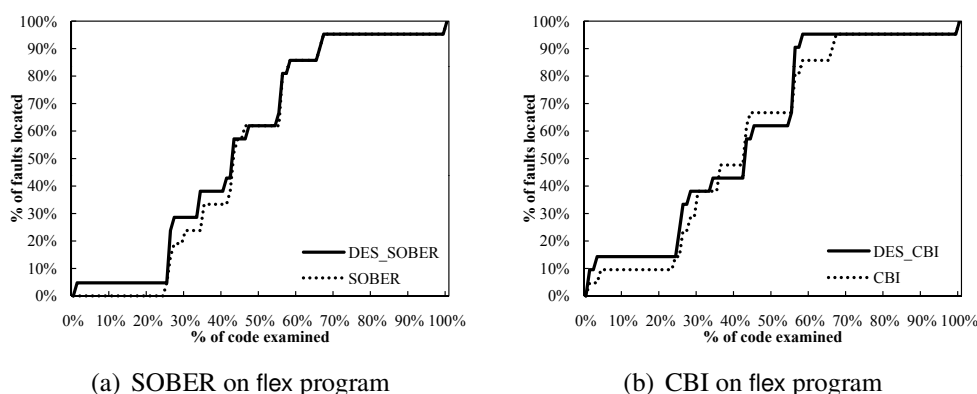
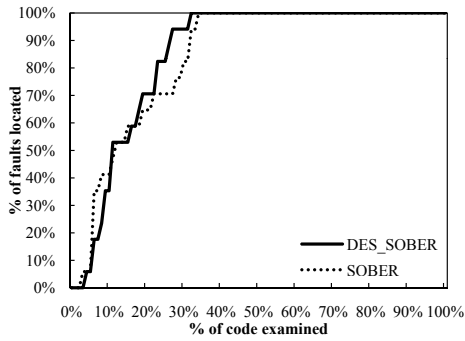


Figure 4.10: Comparisons of DES-enabled techniques with base techniques on flex program.

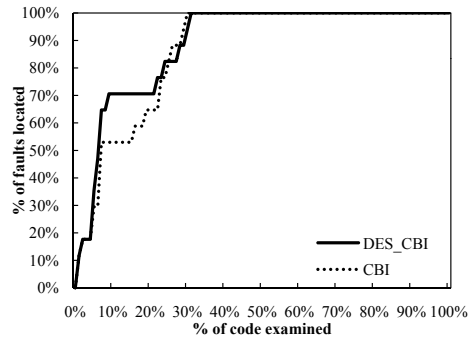
shows advantage over CBI throughout the range from 0 to 100 percent.

For the tcas program, DES_SOBER and DES_CBI obviously perform better than SOBER and CBI, respectively, except that DES_CBI is caught up by CBI when examining more than 60% code.

For the tot_info program, DES_SOBER shows great advantage over SOBER in the code-examination range from 20 to 30 percent. DES_SOBER also shows continuous and steady advantage over SOBER in the range from 50 to 90 percent. In the remaining ranges, DES_SOBER and SOBER perform comparably. At the same time, DES_CBI shows observable advantage over CBI.

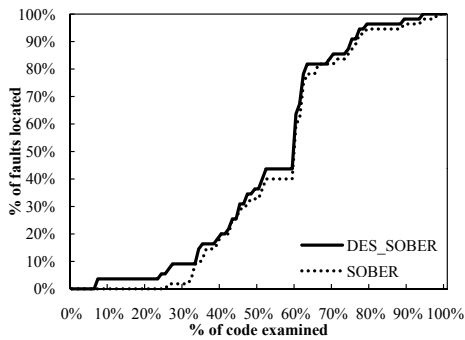


(a) SOBER on grep program

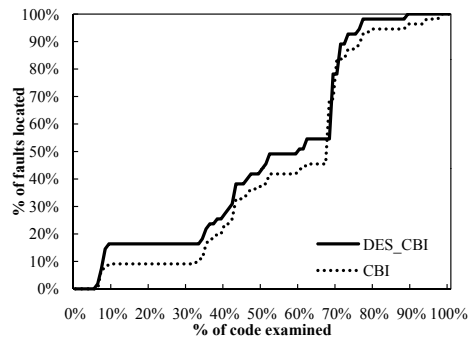


(b) CBI on grep program

Figure 4.11: Comparisons of DES-enabled techniques with base techniques on grep program.



(a) SOBER on gzip program



(b) CBI on gzip program

Figure 4.12: Comparisons of DES-enabled techniques with base techniques on gzip program.

We next move to the flex program. DES_SOBER outperforms SOBER in the code-examination range of 0 to 45 percent, after which they show comparable effectiveness. However, for the same program flex, neither CBI nor DES_CBI shows consistent advantage over each other. CBI is more effective than DES_CBI in the code-examination range of about 35 to 55 percent. In other ranges, DES_CBI is more effective than CBI. On average, they perform comparably to each other.

For the grep program, DES_CBI noticeably outperforms CBI, while there is no obvious difference between DES_SOBER and SOBER. In the first 10 percent code-examination range, SOBER locates more faults than DES_SOBER, but is caught up when examining 10 to 20 percent of the code. For both DES_SOBER



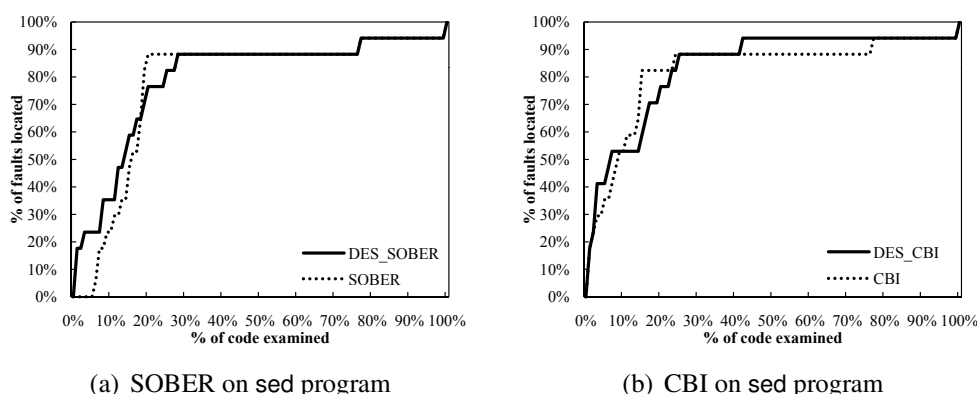


Figure 4.13: Comparisons of DES-enabled techniques with base techniques on `sed` program.

and SOBER, all the faults in the faulty versions of `grep` can be located when examining up to 40 percent of the code. In short, their effectiveness is also comparable.

For the `gzip` program, both DES_SOBER and DES_CBI locate more faults than SOBER and CBI, respectively, in the entire code-examination range.

The comparison results for the `sed` program are like those for the `extsfflex` program. DES_SOBER shows an observable advantage over SOBER, while neither DES_CBI nor CBI shows steady advantage over each other. CBI catches up with DES_CBI only in the code-examination range of 10 to 30 percent. DES_CBI always locates more faults than CBI in other code-examination ranges.

In summary, we observe that, on average, the DES-enabled techniques are comparable to, if not more effective than, their respective base techniques for the programs under study.

Answering RQ1: Is DES effective?

From the results of the Siemens suite of programs (Figures 4.5 to 4.9), we have just observed that the DES-enabled techniques are at least comparable to their base counterparts. However, the Siemens programs are small in size. To further generalize our findings, we have also studied the DES approach on four UNIX utility programs. The results are similar. Since both SOBER and CBI are deemed as effective techniques in previous studies [77], we can, therefore, answer the first research question — the DES approach *is* effective.

Program	Mean relative improvement in effectiveness	Stdev of relative improvement in effectiveness
print_tokens (2 programs)	145%	561%
replace	38%	170%
schedule (2 programs)	94%	377%
tcas	30%	221%
tot_info	12%	132%
flex	-3%	31%
grep	22%	96%
gzip	-4%	22%
sed	-10%	94%
Weighted average	24%	119%
Unweighted average	35%	189%

Table 4.3: Statistics on relative improvements in effectiveness.

Answering RQ2: Is DES better?

Our intuitive observation above, drawn from Figures 4.5 to 4.13, is that the DES approach is comparable to, if not more effective than, the respective base fault-localization technique. We are interested in finding out whether, on average, there is significant advantage in using the DES-enhanced fault-localization techniques over the base techniques.

To do that, we compare for each program the relative improvements in effectiveness of the DES-enabled versions with respect to the base techniques, as shown in Table 4.3. For each program having n faulty versions, we use C_i , DC_i , S_i , and DS_i to represent the t-scores of CBI, DES_CBI, SOBER, and DES_SOBER for the i -th faulty version. We compute $(C_i - DC_i)/C_i$ and $(S_i - DS_i)/S_i$ to estimate the relative improvements in effectiveness when the respective techniques are DES-enabled. We then calculate the mean and standard deviation for the full set of these values (that is, $\{(C_1 - DC_1)/C_1, (S_1 - DS_1)/S_1, (C_2 - DC_2)/C_2, (S_2 - DS_2)/S_2, \dots, (C_n - DC_n)/C_n, (S_n - DS_n)/S_n\}$). We note that each mean and standard deviation are averaged over both DES_SOBER and DES_CBI. From the table, we observe that in 8 programs out of 11, the mean effectiveness of the DES-enabled techniques outperforms that of the respective base techniques.

We also show the weighted averages and unweighted averages for these statistical parameters. The former means averaging the statistical parameters



Program	CBI	SOBER
print_tokens (2 programs)	1.04×10^{-15}	2.67×10^{-10}
replace	2.84×10^{-9}	1.89×10^{-16}
schedule (2 programs)	9.80×10^{-11}	3.00×10^{-3}
tcas	4.17×10^{-10}	3.90×10^{-15}
tot_info	1.79×10^{-13}	4.86×10^{-8}
flex	3.18×10^{-4}	4.00×10^{-11}
grep	2.08×10^{-4}	9.08×10^{-2}
gzip	1.57×10^{-27}	4.76×10^{-26}
sed	2.13×10^{-2}	2.23×10^{-2}

Table 4.4: p-values of U-tests on Siemens programs and UNIX programs.

(means or standard deviations) of each program with weights equal to the number of faulty versions of that program. The latter means directly averaging the statistical parameters for each program. In either case, there is, on average, at least a relative increase of 24% in effectiveness by the DES-enabled versions with respect to base techniques SOBER and CBI. However, the effectiveness improvements are not uniform.

Since, on average, there are effectiveness improvements from a base technique to its DES-enabled version, we want to know whether such improvements are statistically significant. We would like to find out the answer to the following hypothesis:

“ H_0 : Does a technique enabled with the evaluation sequence approach have no significant difference from the base technique?”

If the answer is false, we are confident that the DES approach is significantly different from the base fault-localization technique. Considering our previous observation that the DES approach, on average, improves its base version, we may then regard a DES approach as significantly more effective than its base fault-localization technique.

We perform two-tailed Mann-Whitney U-tests to compare the DES-enabled techniques with the corresponding base techniques with respect to every *individual* subject program. The p-values for hypothesis testing on the programs are listed in Table 4.4.

From the results, we observe that all but one of the p-values are smaller than 0.05, which indicates that the null hypothesis can be successfully rejected at the 5% significance level. (The only exception is the Mann-Whitney U-test between DES_SOBER and SOBER on grep, which has a p-value of 0.09.) In conclusion,



the test result of our null hypothesis H_0 implies that DES-enabled techniques are significantly more effective than their base counterparts. Therefore, our answer to RQ2 is that DES-enabled techniques *are* significantly more effective than their respective counterparts. The answer to RQ2 also confirms that short-circuit evaluation rules *do* have significant positive impacts on statistical fault localization.¹

Besides, we also notice that the DES-enabled techniques are marginally less effective than their base counterparts for the gzip and flex programs. One may anticipate that our approach will have more improvements on the base techniques for programs with higher percentages of compound Boolean expressions (as shown in Table 4.2) than for programs with lower percentages of compound Boolean expressions. This is because our fine-grained approach especially improves the ranking accuracy for compound Boolean expressions. On the other hand, we observe that the DES-enabled techniques perform better on small-sized (Siemens) programs than on medium-sized (UNIX) programs. We notice that the average percentage of compound Boolean expressions (with respect to all expressions) in the UNIX programs is higher than that in the Siemens programs. This unexpected discrepancy can be explained as follows: When we analyze the faults in the Siemens and UNIX programs, we find that higher percentages of faults in the Siemens subjects are on or close to predicate statements, whereas only a few faults in UNIX subjects are on or close to predicate statements. For example, 8 out of 10 faults associated with `print_tokens2` are on the Boolean expressions of predicate statements, while only 4 out of 17 faults associated with version 3 of `flex` are on Boolean expressions of predicate statements.

Answering RQ3: Do different evaluation sequences give the same result?

To answer RQ3, we collect the execution statistics of all the evaluation sequences for each Boolean expression in the Siemens suite of programs to calculate the statistical differences between passed and failure-causing test cases. (Owing to our resource limitation, we do not repeat this part of the experiment on the UNIX subject programs.)

We perform a U-test between the evaluation biases for the sets of evaluation sequences over the same predicate in passed and failure-causing test cases. The results of the U-test show that, for 59.12% of the evaluation sequences, there is a significant difference (at the 5% significance level) between the evaluation biases of passed and failure-causing test cases. In other words, 59.12% of the evaluation sequences are useful fault location indicators, while the remaining 40.87% are

¹ We are conservative about the conclusion because it is subject to external threats to validity to generalize the results.



```

/* Original Version v0 */
if(ch == '\n')

/* Faulty Version v9 */
if(ch == '\n' || ch == '\t')

```

Figure 4.14: Code excerpts from versions v0 and v9 of `print_tokens2`.

not useful standalone fault predicates to differentiate failure-causing test cases from passed ones.

The answer to RQ3 is that different evaluation sequences of the same predicate may have different potentials for fault localization.

4.5.5 Discussion

Like existing predicate-based fault-localization techniques, our DES technique is also developed on top of predicate evaluation. Unlike them, however, it works at a finer granularity. All such class of techniques (including ours) use predicates to indicate the neighborhoods in which faults may reside. The effectiveness of these techniques generally depends on the locations of the faults and how well predicates surround such faults. In this section, we will elaborate on why DES-enabled techniques are more effective than the respective base techniques by taking a closer look at two important cases in faulty programs. We will further discuss other factors that may affect the performance of our techniques. We will also analyze the time complexity and study the empirical performance overheads when applying our technique.

Case 1: Fault on compound predicate

We discuss a fault on a compound predicate in the first case study. The fault is taken from faulty version v9 of the `print_tokens2` program. It is in a decision statement on line 218. The code fragments of the original version and the faulty version are shown in Figure 4.14.

This fault is caused by adding a Boolean expression headed by an “or” operator to the end of the original compound predicate. The fault will be activated only if the original predicate is evaluated to be *false* and the extra Boolean expression is evaluated to be *true* (that is, only if the short-circuit evaluation sequence of the resultant expression is $\langle false, true \rangle$).

DES-enabled techniques divide test cases into two groups, namely, test cases that exercise the evaluation sequence $\langle false, true \rangle$ (thus, triggering the fault that



```

/* Original Version v0 */
ap1: return sum * exp(-x + a * log(x) - LGamma(a))

/* Faulty Version v8 */
ap1: return sum * exp(x + a * log(x) - LGamma(a))

```

Figure 4.15: Code excerpts from versions v0 and v8 of tot.info.

```

cp2: for (p=line; *p != '\0' && isspace((int) *p); ++p)
cp3: if (rdf <= 0 || cdf <= 0)

```

Figure 4.16: Code excerpts from versions v0 and v8 of tot.info.

leads to a program failures), and test cases that do not trigger the fault. As a result, this evaluation sequence is ranked as highly fault-relevant and its corresponding predicate is deemed to be highly related to the program failure. In our experiment, the rank of the faulty predicate is 10 by DES_SOBER and 11 by DES_CBI.

For the corresponding base techniques, however, test cases with evaluation sequences $\langle true \rangle$ and $\langle false, true \rangle$ have been mixed up and treated as similar. As a result, the faulty predicate is perceived by base techniques as less fault-relevant than by DES-enabled techniques. In our experiment, the rank of the faulty predicate is 56 by SOBER and 218 by CBI.

From this case study, we see how a fine-grained analysis technique enables more precise fault localization.

Case 2: Fault on atomic predicate

Let us further focus on a second case, where a fault is seeded on a predicate having an atomic Boolean expression. Specifically, we take this fault from faulty version v8 of the tot_info program. It is a computational fault seeded to an atomic predicate on line 201, as shown in Figure 4.15. For ease of reference, we call this predicate ap_1 .

The whole faulty version includes 46 predicates, only two of which contain compound Boolean expressions. We refer to the first one (on line 57) as cp_2 and the second one (on line 308) as cp_3 , as listed in Figure 4.16. In this example, we use ap_1 to denote an atomic predicate, and cp_2 and cp_3 to denote two compound predicates.

For each of the other 44 atomic Boolean expressions (including ap_1), both CBI and DES_CBI give the same ranking score. The rationale is that the predicates are atomic, and hence there is no possibility of a short-circuit evaluation.



However, CBI gives ranks of 46 and 25 to predicates cp_2 and cp_3 , respectively, while our DES_CBI technique gives ranks of 46 and 45, respectively. This is because these two are compound predicates and CBI and DES_CBI may generate different ranking scores (and hence different ranks) for them. Finally, the faulty predicate ap_1 is ranked as 24 by CBI, and ranked as 23 by DES_CBI. Thus, DES_CBI make a more correct assessment that cp_2 and cp_3 are less fault-relevant than ap_1 , whereas CBI mistakenly gives higher suspiciousness to cp_3 than ap_1 .

A similar phenomenon is observed for SOBER and DES_SOBER. SOBER gives ranks of 20 and 7 to predicates cp_2 and cp_3 , respectively, while DES_SOBER gives ranks of 38 and 41, respectively. For each of the other 44 atomic predicates (including ap_1), both SOBER and DES_SOBER generate the same relative ranking. The faulty predicate ap_1 is ranked as 22 by SOBER and 20 by DES_SOBER. Thus, DES_SOBER make a more correct assessment that cp_2 and cp_3 are less fault-relevant than ap_1 , whereas CBI mistakenly gives higher suspiciousness to cp_2 and cp_3 than ap_1 .

In cases where faults are on atomic predicates, there may also exist other predicates that contain compound Boolean expressions. From our previous case study about faults on compound Boolean predicates, we know that DES-enabled techniques may give more accurate ranking results on these compound predicates than SOBER and CBI do. Thus, the noise (possible inaccurate ranking results) from other compound predicates can be reduced. The present case study confirms that DES-enabled techniques may produce a more accurate ranking of predicates even if the faulty predicate is atomic.

Time complexity, actual time-cost, and other discussions

Let p_1, p_2, \dots, p_m be the Boolean predicates of the program, and k_1, k_2, \dots, k_m be the numbers of atomic Boolean expressions in the respective predicates. Suppose the time complexity for applying a base technique to investigate one predicate is O_{base} . The time complexity for applying a base technique to the program will be $O(O_{base} \times m)$. The time complexity of the corresponding DES-enabled technique will then be $O(O_{base} \times \sum_{i=1}^m k_i)$. This is because a DES-enabled technique uses the same algorithm as the base technique, and the only difference is that the DES-enabled technique works on evaluation sequences while the base technique works on predicates. Thus, for each evaluation of a predicate, in the worst case, it will evaluate all the atomic components of the predicate, and call an invocation of the base algorithm every time.

Thus, the time complexity of applying DES on a base technique is higher



than that of the base technique. It is easy to figure out that the increase of the time complexity from the base technique to its DES-enabled version is $\frac{1}{m} \sum_{i=1}^m k_i$. This number is the average number of atomic expressions inside the Boolean expressions in the program. We are confident that it is not a large number in realistic programs. For instance, this number is always less than 5 in the Siemens and UNIX programs used in our experiment.

In addition, the data structure of the evaluation sequence needs to be kept during evaluation. What if we translate each Boolean expression into binary code (or lower level representation) and perform a statement-level fault-localization technique on each assembly instruction? Using such a transformation, every atomic component in a compound Boolean expression can be considered, say, as a complete assembly instruction, and the construction of evaluation sequences can be avoided. However, the executions of such instruction statements are not independent of one another, and hence separately estimating their suspiciousness from their execution status may not be accurate. One may further argue to correlate a set of predicates (or statements) to improve the effectiveness of fault identification. We argue, however, that finding such a set of predicates is the exactly basic idea behind our approach. An evaluation sequence contains the information of the legitimate value combinations of atomic predicates that developers compose in the code. We believe that it is a natural and objective criterion to find out such a set of correlating predicates in programs.

What if a technique uses the full combination of truth values of each atomic Boolean expression, but does not consider the evaluation sequences? Suppose $b_1 \oplus b_2 \oplus \dots \oplus b_n$ (where \oplus stands for a logical operator) is a compound Boolean expression. Since each atomic Boolean expression b_i may have a truth value of either *true* or *false*, the full combination of truth values of these n atomic Boolean expressions is a set of 2^n elements. The time complexity of such a proposal will be $O(O_{base} \times 2^n)$, and some value combinations are very likely to be illegitimate in actual program executions. Consider, for instance, a Boolean expression “ $p \neq \text{null} \ \&\& \ p[0] \neq \text{null}$ ”. The value combination of $\langle \text{false}, \text{true} \rangle$ cannot appear in any actual program execution owing to the short-circuit evaluation logic in the C language. Compared with the fault indicators above, evaluation sequences of predicates are natural, objective, and effective program entities to extract dynamic features for fault localization.

An empirical study of the actual performances of the DES-enabled techniques compared with those of the respective base techniques is listed in Table 4.5. The actual time-cost in each step is small enough for practical applicability.



Time-cost	DES		Base
	Siemens Programs	UNIX Programs	Siemens/UNIX Programs
Instrumentation	comparable to gcc compilation time		
Exe. statistics collection	about 1/10 of program execution time		
Statement ranking	0.1×10^{-3} to 15.7×10^{-3} s	16.8×10^{-3} to 120.7×10^{-3} s	less than 0.1 s
Locating fault using generated ranking list	7.0×10^{-6} to 50.0×10^{-6} s	18.0×10^{-6} to 823.0×10^{-6} s	less than 0.1 s

Table 4.5: Timing statistics in the experiment.

4.5.6 Threats to Validity

We briefly summarize below the threats to validity in our controlled experiment.

Construct Validity

In this experiment, construct validity is related to the platform dependence issues when using the Siemens suite of programs in SIR [40]. Since every program in SIR has a fault matrix file to specify the test verdict of each test case (that is, whether it is a passed or failure-causing test case), we also create a fault matrix file for our test results and carefully verify each test verdict against the corresponding one supplied by SIR. We observe that there are only minor differences in test verdicts between the two fault matrix files. We have thoroughly verified our setting, and believe that the difference is due to platform dependence issues.

Internal Validity

In this experiment, internal validity is related to the risk of having confounding factors that affects the observed results. Following [76], in the experiment, each technique uses all the applicable test cases to locate fault-relevant predicates in each program. The use of a test suite with a different size may give a different result [76]. Evaluations on the impact of different test suite sizes on our technique would be welcome. Another important factor is the correctness of our tools. Instead of adopting existing tools used in the literature, we have implemented our own tools in C++ for the purpose of efficiency. To avoid errors, we have adhered to the algorithms in the literature and implemented and tested our tools carefully. To align with previous work, we use the t-score metric to compute the results of this experiment. The use of other metrics may produce different results.



Internal validity is also related to any affecting factors we may or may not have realized. As shown in Section 4.5.4, we have listed the related statistics and explained the reason why our technique appears to be more effective on the small-sized subject programs than the medium-sized subject programs. There may be other implicit factors that may affect the effectiveness of our technique and other predicate-based techniques.

External Validity

We use the Siemens suite and four UNIX utility programs in the experiment to verify the research questions because they are commonly used by researchers in testing and debugging studies with a view to comparing different work more easily. Further applications of our approach to medium-to-large-sized real-life programs would strengthen the external validity of our work. Each of the faulty versions in our subject programs contains one fault. Despite the competent programmer hypothesis, real-life programs may contain more than one fault. Although [77] have demonstrated that predicate-based techniques can be used to locate faults in programs that contain more than one fault, their effectiveness in this scenario is not well discussed. We will address this threat in future work.

4.6 Summary

Following current popular trend of statistical fault localization, we have explored a better way to measure and rank predicates with respect to fault relevance. We observed that the fault-localization capabilities of various evaluation sequences of the same Boolean expression are not identical. Because of short-circuit evaluations of Boolean expressions in program execution, different evaluation sequences of a predicate may produce different resultant values. This inspired us to investigate the effectiveness of using Boolean expressions at the evaluation sequence level for statistical fault localization. The experiment on the Siemens programs and UNIX utility programs showed that our approach is promising.

The major contribution of this chapter is twofold. (i) We provide the first set of experimental results regarding the effect of short-circuit evaluations on statistical fault localization. (ii) We show that short-circuit evaluation has a significant impact on the effectiveness of predicate-based fault-localization techniques.



Chapter 5

Slope: Statistical Fault Localization via Failed Program Executions

In the chapter before previous, we investigate how to capture the propagations of infected program states and locate faults that cannot be effectively located by previous existing techniques. In the previous chapter, we also investigate how to address the issue of short-circuit evaluation rule and conduct finer-grained fault localization to gain better effectiveness. However, we notice that all previous techniques rely on passed executions to locate fault, and their effectiveness is greatly reduced in environment where passed executions are unavailable, for example, a debugging driven by bug reports.

In this chapter, we develop a technique that can work in the absence of passed executions, to locate fault. We first introduce the background and illustrate how previous statistical fault-localization techniques rely on both passed and failed executions to locate fault, and then use a motivating example to demonstrate how to locate fault with solely failed program executions. After that, we elaborate on our model – Slope, and then use empirical study to validate the effectiveness of our technique.

5.1 Background

To help effectively locate faults in a faulty program, the common basic idea of many existing statistical fault-localization techniques is to estimate the fault-suspiciousness of each statement (or other program elements) of the program first. They compare the execution statistics on program spectra [91] between the program's passed execution(s) and failed execution(s), and rank statements (or other program elements) according to their fault-suspiciousness scores. For instance, if no passed execution exercises a particular statement and yet every



failed execution exercises it, Tarantula [68] deems the statement to be suspicious (with a score of 1) and ranks it higher than another statement (e.g., with a score of 0). As such, even though failed executions or failure-causing test cases can be directly provided by bug reports [88] or reported by user application, following previous work (e.g., [76]), developers should find passed executions to pair up with the failed executions to locate fault effectively.

Such an activity can be a painstaking to developers. For example, modern software is frequently equipped with failure-feedback bug report mechanism. Such bug reports may contain versatile realistic failed executions [88]. Developers may require constructing their own passed executions that can be usefully to compare with the given failed executions. Thus, fault localization via such technique cannot effectively be conducted until passed executions are available. It may introduce a delay to debug the programs. Worst still, the effectiveness of such a technique depends heavily on the quality of passed executions that may exhibit coincidental correctness [106] or the other problems, which may hardly be known until the causing faults are located.

What if such a fault-localization model, which contrast the execution information of passed and failed executions to locate fault, may also work with solely failed executions? Developers can immediately apply a technique based on such a model as long as they have obtained failed executions. It provides developers a time window to both locate faults and generate additional passed executions. Once passed execution has been obtained, developers can apply them to a technique of the same model to locate faults even more effectively. On the other hand, our experiment reported in this thesis shows that merely using failed executions, existing techniques are ineffective. Thus, developers may require switching between techniques developed on top of diverse ideas between the two phases. It may overload the minds of developers.

5.2 Motivation

Figure 5.1 shows two code fragments (with line numbers on the left) excerpted from the faulty version. The upper one (L115–L124) shows an if-structure that evaluates a compound predicate consisting of a character checking function (at L115) and a boundary condition (at L117). If this predicate is evaluated to be *true*, the variable `dest` will be modified by the `addstr` function (at L121); otherwise, `dest` will remain unchanged. The lower code fragment (L495–L502) outputs the characters in a string.

In the upper code fragment, the predicate of the if-statement is faulty because an operand of the expected version of the predicate is missing. Figure



5.1 also shows the missing part (L116). Note that the faulty statement is more likely to be evaluated to be *true* than the expected version, so that an execution has a higher chance of exercising L121 than the expected version. As a result, the variable `dest` may contain an incorrect value. We further observe that the predicate in the faulty statement may produce a result different from its expected version only when the missing function call to `isalnum()` returns *false*, which in turn depends partially on its input parameter `src`. It is generally hard to predict the string content statically. The chance of producing an incorrect decision value by the faulty statement is thus hard to know in general. Despite the absence of further information, we postulate that *the more frequently the faulty statement has been executed, the higher will be the chance that it produces an incorrect decision value leading to a failed execution.*

L115	<code>if ((isalnum(src[*i - 1]))</code>
L116	<code>/* missing code "&& (isalnum(src[*i + 1]))" */</code>
L117	<code>&& (src[*i - 1] <= src[*i + 1]))</code>
L118	<code>{</code>
L119	<code>for (k = src[*i-1]+1; k<=src[*i+1]; k++)</code>
L120	<code>{</code>
L121	<code>junk = addstr(k, dest, j, maxset);</code>
L122	<code>}</code>
L123	<code>*i = *i + 1;</code>
L124	<code>}</code>

L495	<code>if ((m >= 0) && (lastm != m)) {</code>
L496	<code>putsub(lin, i, m, sub);</code>
L497	<code>lastm = m;</code>
L498	<code>}</code>
L499	<code>if ((m == -1) (m == i)) {</code>
L500	<code>fputc(lin[i], stdout);</code>
L501	<code>i = i + 1;</code>
L502	<code>}</code>

Figure 5.1: Faulty version v10 of `replace`

To verify the above hypothesis, we execute the faulty version over every one of the 5542 test cases of `replace` provided by SIR. For every statement s_i , we record its execution count c_i for each program execution. For each value c of execution count, we tally the number of program executions $N_i(c)$ having a count $c_i = c$, and calculate the fraction of failed executions among the $N_i(c)$ executions (see Definition 5.3.1 for *failing rate* in Section 5.3).



For comparison purpose, we choose three other statements, namely L121, L497, and L500 to examine. We observe from Figure 5.1 that L121 is closer to the faulty statement in the program (in terms of the number of lines in the source code) than either L497 or L500. Developer may find L121 suspicious because it may modify the variable `dest` wrongly as the result of the faulty statement. On the other hands, we have inspected the source code to ensure that the fault is not related to the logic at L497 or L500.

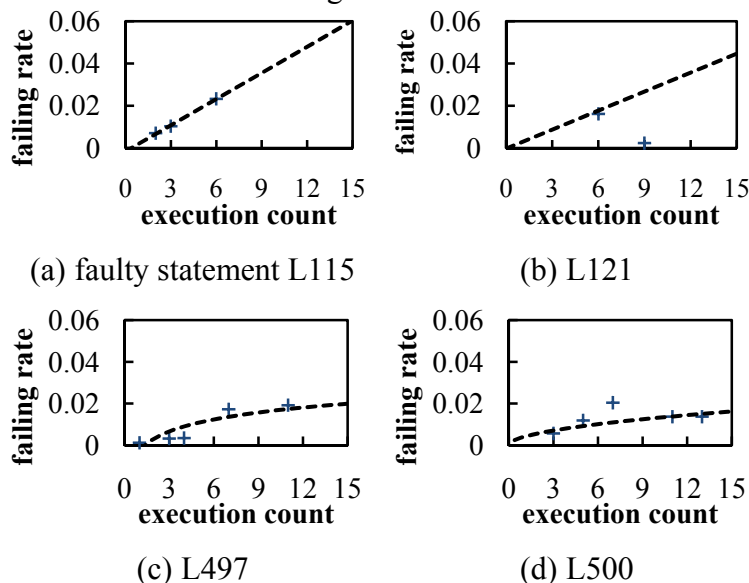


Figure 5.2: Failing rate vs. execution count

We then plot the failing rate against the execution count for the faulty statement and fit the points using a curve. The result is shown in Figure 5.2(a). Similarly, we show such plots for L121, L497, and L500 in Figure 5.2(b), Figure 5.2(c), and Figure 5.2(d), respectively. Owing to the page limit, only parts of the plots are shown; yet the curve in each plot does fit all the points.

We have a few observations from Figure 5.2. Overall, as the execution count increases, the failing rate in Figure 2(a) or Figure 2(b) increases faster than that in Figure 2(c) or Figure 2(d). It preliminarily shows that our intuition on faulty statements holds; furthermore, fault-suspicious statements (such as L121 in Figure 2(b)) appear to have similar properties. On the other hand, for statements least related to a fault, their changes in failing rates with respect to execution counts are not as steep as those of faulty statements or fault-suspicious statements. This motivates us to estimate the fault-suspiciousness of statements by analyzing their execution counts and failing rates. Further, we may perform curve fitting on the points formed by the value of execution

count and that of failing rate, and then use the first derivative of the curve function to measure, for every statements, how steep the changes in failing rates are. On the other hand, the fitting error means how reliable such an analysis can be and how confident we are. Therefore, we decide to use the signal-to-noise ratio [95] as the final ranking metric.

Although this example motivates us to develop a new fault-localization method, we foresee potential challenges. To compare the changes in failing rate with respect to execution count, what is the form of curve that fits the points in [125]? If we follow this example strictly, the computation of failing rates involves the execution counts in both passed and failed executions. How can we finally eliminate the dependency on passed executions in our model so that we can make our new technique also work with solely failed executions as well? In the next section, we are going to elaborate on our model to address the challenges.

5.3 Our Fault-localization Model

5.3.1 Problem Setting

Suppose P is a faulty program represented by a set S of statements $\{s_1, \dots, s_i, \dots, s_n\}$. Let $\{t_1, \dots, t_j, \dots, t_m\}$ be a set of failed executions of P . Further let $\{t'_1, \dots, t'_j, \dots, t'_m\}$ be a set of passed executions of P . We use the term C_{ij} to denote the *execution count* of s_i in t_j . It means the number of times that the statement s_i has been exercised in the failed execution t_j .

Our aim is to develop a technique to construct a *suspicious list* that contains all statements in S sorted in descending order of their suspiciousness. To ease our presentation, we will consistently use the subscript i (including $i1$ or $i2$) to refer to a statement, and the subscript j (including $j1$ or $j2$) to refer to an execution.

5.3.2 Our Observation

Different executions may exercise the same statement for different numbers of times. In other words, $C_{i,j1}$ may be different from $C_{i,j2}$ for $j1 \neq j2$. To facilitate our model development, we first define the concept of *failing rate* as follows.

Definition 5.3.1: The **failing rate** $F_i(c)$ of s_i calculates the fraction of failed executions with respect to all executions (passed or failed) that each exercises the statement s_i exactly c times. Thus:



$$F_i(c) = \frac{|\{t_j | C_{i,j} = c\}|}{N_i(c)} \text{ if } N_i(c) \neq 0; \text{ otherwise undefined.}$$

where $|\{t_j | C_{i,j} = c\}|$ stands for the number of failed executions that each exercises s_i exactly c times, and $N_i(c)$ stands for the number of executions (passed or failed) that each exercises s_i exactly c times. In particular, if no failed execution exercises s_i exactly c times, the value of $F_i(c)$ is also undefined.

Note that the value range of $F_i(c)$ is $[0, 1]$. Take Figure 5.2(b) for illustration: $F_{L121}(6)$ is 0.016 and $F_{L121}(3)$ is undefined.

In Section 5.2 we have preliminarily illustrated that the more frequently a faulty statement has been exercised by a program execution, the more likely will the program execution reveal a failure. Substituting the corresponding terms in Definition 5.3.1, we obtain the following heuristics. *For a faulty statement s_i , the higher the value of c , the higher will be the value of the corresponding $F_i(c)$.* We further observed in Section 5.2 that L121, which is close to the faulty statement, also has such a trend. We deem that $F_i(c)$ can ideally be considered as a discrete monotonic increasing function of c if the statement s_i is faulty or is close to a fault and affected by it (which we called the statement fault-suspicious). In the next section, we will use this heuristics to develop our technique that works in scenario where both passed and failed executions are available. In particular, we will present how we further eliminate $N_i(c)$ to work out another formula that can be used in scenario where only failed executions are available.

5.3.3 Our Model -- Slope

Our model can be explained via a three-stage process: *calibration* stage, *fitting* stage, and an optional *elimination* stage. These stages are shown in Figure 5.3. However, our technique, **Slope**, rely on the result of our model, rather than this process (as shown in Figure 5.3). In the calibration stage, we calibrate the failing rates $F_i(c)$ to create $G_i(c)$ for alleviating the impact of the potential presence of multiple faults in a faulty program on the curve fitting accuracy. In the fitting stage, we use a linear function to fit the data points of $\langle c, G_i(c) \rangle$, for all available value of c . Figure 5.3 also show two scenarios. In scenario where both passed and failed executions are available, we directly use the results of line fitting to calculate the fault-suspiciousness score of each statement. In scenario where only failed executions are available, we continue the elimination stage to eliminate dependence on passed executions and make our model applicable to scenarios where only failed executions are available. By sorting all the statements according to their fault-suspiciousness scores, a



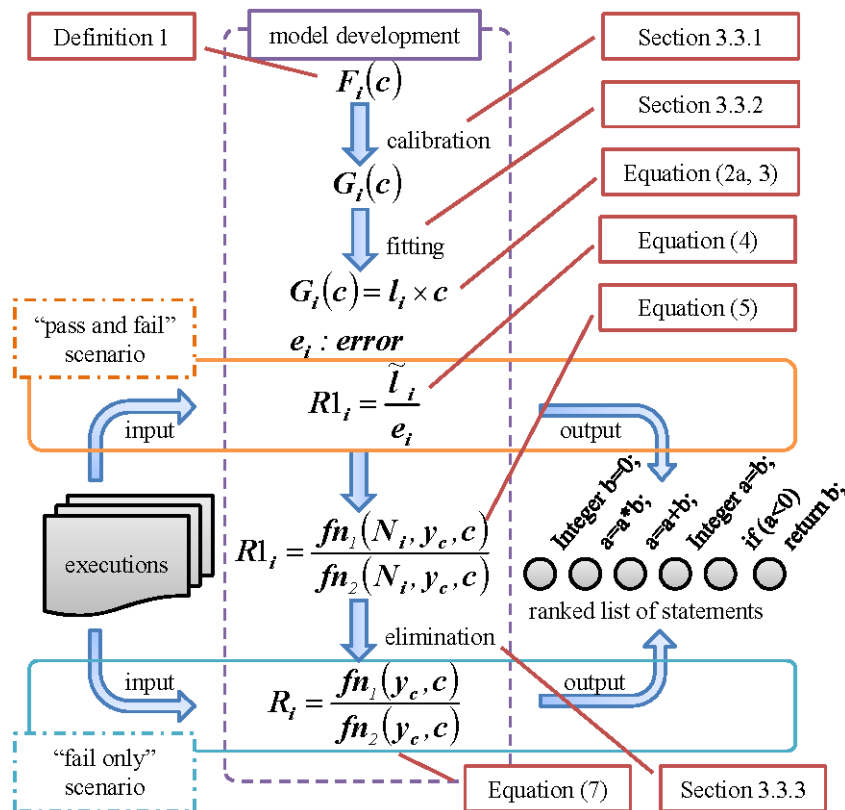


Figure 5.3: Framework of our model

ranked list of statements is constructed. Developers may then consider the statements to look for the faults according to the positions of the statements in the suspicious list, sorted in descending order of their fault-suspiciousness.

Calibration Stage

Let us consider for discussion a scenario in which P contains multiple faults. Suppose a failed execution t_j exercises s_i at least once and exercises other statements as well. In general, if we have no further information, it is difficult to decide whether the failure associated with t_j is caused by executing s_i or by

executing the other statements. In an attempt to reduce the noise caused by other faulty statements of P , let us take a look at $F_i(0)$ first. $F_i(0)$ stands for the portion of failed executions that s_i has not been exercised, and yet failures are observed from associated executions. These failures must be caused by fault(s) other than that in s_i . Our idea is to use $F_i(0)$ to approximate the effect of the overall failing rate of all faulty statements except s_i , and use it to calibrate the failing rate $F_i(c)$ of s_i in terms of

$$G_i(c) = F_i(c) - F_i(0).$$

The calibrated failing rate $G_i(c)$ aims to estimate the “probability” of “*an execution, which exercises a statement s_i for **exactly** c times, leads to a failure*”.

Note that there is no guarantee $F_i(0)$ is less than $F_i(c)$, so the value range of $G_i(c)$ is $[-1, 1]$. Since we are interested in the trend of changing of $F_i(c)$ with respect to the change in c , we perform the linear transformation from $F_i(c)$ to $G_i(c)$, which preserve the trend of changing and has more accurate physical meaning. Note that if $F_i(c)$ is undefined, $G_i(c)$ will be also undefined. In particular, if every execution exercises s_i , $F_i(0)$ is undefined. In such a case, we take $F_i(0) = 0$.

Fitting Stage

By pairing up each c and the corresponding $G_i(c)$ such that $G_i(c)$ is defined, we can create a point $\langle c, G_i(c) \rangle$ in a two-dimensional (2D) space for s_i . Following the heuristics given in Section 5.3.2, for a fault-suspicious statement s_i , $G_i(c)$ should ideally be a discrete monotonic increasing function of c . Therefore, we propose to make use of a monotonic curve to fit these points on such a 2D plane. The details are as follows:

We estimate, in two ways, the probability of “*an execution, which exercises s_i **exactly** c times, does not result in a failure*”. First, we estimate it as $1 - G_i(c)$. The second way is by using the probability that each time (out of a total of c times) of executing s_i does not lead to a failure. Let us denote by p_i the probability of “*an execution, which exercises s_i for only once, leads to a failure*”. This probability can be estimated to be $(1 - p_i)^c$. Equating the two probabilities obtained, we have

$$1 - G_i(c) = (1 - p_i)^c \quad (5.1)$$

Substituting c by x and $G_i(c)$ by $f(x)$, we observe that equation (5.1) is in the format of $f(x) = 1 - (1 - p_i)^x$. From Taylor series [125], we know that



$$l_i = \frac{\sum_{c \in D_i} [\theta_c \cdot G_i(c)]}{\sum_{c \in D_i} [\theta_c^2]} \quad (5.2a)$$

$$\tilde{l}_i = l_i \times \max\{c \mid c \in D_i\} \quad (5.2b)$$

$$e_i^2 = \sum_{c \in D_i} [(G_i(c))^2] - \frac{(\sum_{c \in D_i} [\theta_c \cdot G_i(c)])^2}{\sum_{c \in D_i} [\theta_c^2]} \quad (5.3)$$

$f(x)$ can be approximated by a sum of terms calculated from the values of its derivatives at a single point x_0 .

$$f(x_0) + f^{(1)}(x_0) \frac{x}{1!} + f^{(2)}(x_0) \frac{x^2}{2!} + \dots + f^{(n)}(x_0) \frac{x^n}{n!} + \varepsilon_n(x)$$

where $\varepsilon_n(x)$ is the error term [125]. We propose to use the sum of the first two terms $f(0) + f^{(1)}(0) x$ to approximate the value of the function $f(x)$ because these two terms have captured the basic idea of our heuristics presented in Section 5.3.2. Thus, we have:

$$\begin{aligned} G_i(c) &\approx G_i(0) + G_i^{(1)}(0) \frac{c}{1!} \\ &= [1 - (1 - p_i)^0] - \log(1 - p_i) \cdot (1 - p_i)^0 \cdot c \\ &= -\log(1 - p_i) \cdot c \end{aligned}$$

Recalling that $G_i(c)$ is designed to fit the points of $\langle c, G_i(c) \rangle$ with respect to s_i , for different values of c , and we have chosen $G_i(c)$ to be a linear function. We may further perform linear fitting on these defined points, and use the slope of the fitted line to find out the value of p_i to stand for the fault-suspiciousness of s_i .

We further recall from mathematics that, for $p_{i1}, p_{i2} \in [0, 1]$, $p_{i1} > p_{i2}$ whenever $-\log(1 - p_{i1}) > -\log(1 - p_{i2})$; $p_{i1} < p_{i2}$ whenever $-\log(1 - p_{i1}) < -\log(1 - p_{i2})$; and $p_{i1} = p_{i2}$ whenever $-\log(1 - p_{i1}) = -\log(1 - p_{i2})$. This motivates us to define l_i to be $-\log(1 - p_i)$ to simplify the algebra, and use the slope of the line directly to provide information of the fault-suspiciousness of s_i . Substituting $-\log(1 - p_i)$ by l_i , we have:

$$G_i(c) = l_i \cdot c$$

Note the value range of l_i is $[-1/c, 1/c]$. Since l_i means in geometry the slope of a line, we name our technique as **Slope**.

We apply least square analysis [125] to minimize the error in line fitting. The standard results of linear fitting are given as equations (5.2a) and (5.3). l_i is the fitting result, e_i ($e_i > 0$) is the fitting error, and D_i is the input domain of



c for s_i (that is, $D_i = \{c \mid G_i(c) \text{ is defined}\}$). After fitting the points, we further normalize l_i to \tilde{l}_i in equation (5.2b). Our reason is that the execution counts of different statements may not be the same, even though the calibrated failing rate has already been in the range $[0, 1]$. (For example, L123 in Figure 5.2 may be executed much less frequently than L121.) Because l_i is the ratio of $G_i(c)$ over c , and the range of c for s_i is from 0 to $\max\{c \mid c \in D_i\}$, where $\max\{c \mid c \in D_i\}$ means the maximum value among all c in D_i , we multiply l_i by $\max\{c \mid c \in D_i\}$ to calculate the normalized value of l_i .

For each s_i , we have worked out two parameters, namely \tilde{l}_i and e_i . Here \tilde{l}_i indicates the *mean* strength of the signal for the concerned statement being suspicious to be faulty; and e_i represents the standard deviation of the fitting data, indicating the error of conformance when using our model to fit the data points to a line. In our model, we intend to use \tilde{l}_i to estimate the fault-suspiciousness of the statement s_i (the higher the better), and e_i to measure the reliability of \tilde{l}_i (the lower the better). Inspired by the standard notion of signal-to-noise ratio (which is defined as a mean over a standard deviation), we design equation (5.4) accordingly.

$$R1_i = \frac{\tilde{l}_i}{e_i} \quad (5.4)$$

We further substitute \tilde{l}_i and e_i by their formulas (equations (5.2b) and (5.3), respectively) in equation (5.4) to obtain equation (5.5), and denote $\{t_j \mid C_{i,j} = c\}$ by y_c to simplify the presentation. In our technique, **Slope**, we use equation (5.5) to determine the fault-suspiciousness of the statement s_i . For the ease of reference, we refer to $R1_i$ in equation (5.5) as the ranking score of the statement s_i . Note that the value range of $R1_i$ is $[-\infty, +\infty]$. The higher the value of $R1_i$, the more fault-suspicious will be the statement s_i .

Elimination Stage (Optionally used when Passed Executions are Unreliable to Use or Unavailable)

$$R1_i = \frac{\sum_{c \in D_i} \left[c \cdot \left(\frac{y_c}{N_i(c)} - \frac{y_0}{N_i(0)} \right) \right] \times \max\{c \mid c \in D_i\} / \sum_{c \in D_i} [c^2]}{\sqrt{\sum_{c \in D_i} [c^2] \cdot \sum_{c \in D_i} \left[\left(\frac{y_c}{N_i(c)} - \frac{y_0}{N_i(0)} \right)^2 \right] - \left(\left(\sum_{c \in D_i} \left[c \cdot \left(\frac{y_c}{N_i(c)} - \frac{y_0}{N_i(0)} \right) \right] \right)^2 / \sum_{c \in D_i} [c^2] \right)}} \quad (5.5)$$



$$R_i = \frac{\sum_{c \in D_i} \left[c \cdot \left(\frac{y_c}{\bar{N}_i} - \frac{y_0}{\bar{N}_i} \right) \right] \times \max\{c \mid c \in D_i\} / \sum_{c \in D_i} [c^2]}{\sqrt{\sum_{c \in D_i} [c^2] \cdot \sum_{c \in D_i} \left[\left(\frac{y_c}{\bar{N}_i} - \frac{y_0}{\bar{N}_i} \right)^2 \right] - \left(\left(\sum_{c \in D_i} \left[c \cdot \left(\frac{y_c}{\bar{N}_i} - \frac{y_0}{\bar{N}_i} \right) \right] \right)^2 / \sum_{c \in D_i} [c^2] \right)}} \quad (5.6)$$

$$R_i = \frac{\sum_{c \in D_i} [c \cdot (y_c - y_0)] \times \max\{c \mid c \in D_i\} / \sum_{c \in D_i} [c^2]}{\sqrt{\sum_{c \in D_i} [(y_c - y_0)^2] - \left(\left(\sum_{c \in D_i} [c \cdot (y_c - y_0)] \right)^2 / \sum_{c \in D_i} [c^2] \right)}} \quad (5.7)$$

From equation (5.5), we observe that the calculation of $R1_i$ requires $N_i(c)$, which in turn requires the execution counts of passed executions. When there are only failed executions (such as in the scenario described in Section 1 to support an earlier start of fault localization), we continue the next stage to eliminate the parameter $N_i(c)$ to make a formula that does not rely on passed executions or passed test cases.

To eliminate this parameter $N_i(c)$ from Equation (5.5), for each statement s_i , we use \bar{N}_i as the estimate of $N_i(c)$ as follows, where \bar{N}_i is the mean value of $N_i(c)$ for all c for the statements s_i .

$$\bar{N}_i = \frac{1}{|D_i|} \sum_{c \in D_i} N_i(c)$$

After substituting each $N_i(c)$ in equation (5.5) by \bar{N}_i , we obtain equation (5.6). After simple mathematical manipulation, \bar{N}_i can be eliminated from equation (5.6) and we get equation (5.7). Although in previous model development steps (e.g., Equation (5.5) and (5.6)), the computation of $N_i(c)$ related to passed executions, now, equation (5.7) is fully independent of any passed execution parameter.

5.3.4 Dealing with Expectance Cases

For an executable statement s_i with only one sample point, it means that all failed executions exercise that statement. In such a case, equation (5.7) cannot be directly applied, and we assign its ranking score R_i as $+\infty$ (or the largest value supported by system). When the value of the denominator in equation (5.7) is zero, the equation is undefined, in which case we assign a ranking score of $-\infty$ (or the largest negative value supported by system). For a statement s_i with no sample point, which means s_i has not been exercised in



any failed executions, we always append such statements to the end of the resultant list of statements since such statements have least possibility to be related to faults.

The technique first sorts statements in decreasing ranking scores ($R1_i$ is used when both passed and failed executions are available, R_i is used when only failed executions are available) and resolves tie cases by next sorting in decreasing order of the numerator of equation (5.7) (indicating the slope of the linear curve). For statements in tie cases that still cannot be resolved, Slope “assigns them a rank as the sum of the number of the tied statements and the number of statements ranked before them” [112].

5.4 Empirical Evaluation

5.4.1 Subject Programs

We use seven Siemens programs and four UNIX tool programs as our experiment subjects. They all have been used in previous work (such as [76][116]) to evaluate existing fault-localization techniques. We downloaded them from SIR [40]. There are several versions for each subject program. Each version contains one or more faults. SIR also supplies a test pool for each subject program.

Table 1 shows the statistics of these subject artifacts. Take `flex` as an example. The real-life versions used are in the range of `flex-2.4.7` to `flex-2.5.4`. In total, 81 faults are associated with `flex`, and 18 of them are finally chosen in our experiment. (The selection strategy will be described later.) For these 18 faulty versions, the number of lines of executable statements of each version is in the range of 8571 to 10124. SIR provides a test pool with 567 test cases for `flex`. In total, we use 122 Siemens program faulty versions and 64 UNIX faulty versions in our single-fault experiment, and use 20 faulty versions of UNIX programs in our multi-fault experiment because UNIX programs have medium scales and are more applicable to seed multiple faults.

5.4.2 Peer Techniques



Table 5.1: Statistics of subject programs

Name of program	Version in real-life	# of faults	# of executable LOC	# of applicable versions (single-fault / multi-fault)	Size of test pool
print_tokens	unknown*	7	341-342	5 / 0	4130
print_tokens2	unknown*	10	350-354	10 / 0	4115
replace	unknown*	32	508-515	30 / 0	5542
schedule	unknown*	9	291-294	6 / 0	2650
schedule2	unknown*	10	261-263	8 / 0	2710
tcas	unknown*	41	133-137	40 / 0	1608
tot_info	unknown*	23	272-274	23 / 0	1052
flex	2.4.7-2.5.4	81	8571-10124	18 / 4	567
grep	2.2-2.4.2	57	8053-9089	17 / 6	809
gzip	1.1.2-1.3	59	4035-5159	13 / 4	217
sed	1.18-3.02	25	4756-9289	16 / 6	370
Total (single-fault / multi-fault):				186 / 20	

* These real-life version numbers cannot be found in SIR.

We choose four techniques, namely Jaccard [1], Ochiai [1], Tarantula [68], and SBI [112] to compare with our technique. We do not select CBI or SOBER because in our previous studies [120], we have empirically found that their effectiveness on the UNIX programs `flex` and `gzip` are much poorer than those of Jaccard, Tarantula, and SBI. Moreover, the two predicate-level techniques need to be evaluated using another metric (t-score [90]).

Tarantula [68] counts the chance that a statement in a program has been exercised by failed executions as well as the chance that the statement has been exercised by passed executions. It then uses the ratios of the former over the sum of the former and the latter to estimate how much the statement is fault-suspicious. Finally, all statements are sorted in a list according to such computed fault-suspiciousness, and all the non-executable statements are appended to the list. We show Tarantula's ranking formula, $R_{Tarantula}(s_i)$ below, where m and m' mean the number of failed executions and the number of passed executions, respectively. $failed(s_i)$ and $passed(s_i)$ are the number of failed executions and the number of passed executions, respectively, that exercise s_i at least once. Tie cases are solved by sorting the statements in descending order of confidence $Conf(s_i)$ [112].

$$R_{Tarantula}(s_i) = \frac{failed(s_i)/m}{failed(s_i)/m + passed(s_i)/m'}$$



$$Conf(s_i) = \max \{ failed(s_i)/m, passed(s_i)/m' \}$$

CBI [75] aims to identify a program's fault-relevant predicates [14]. It computes the probability that a predicate is evaluated to be *true* in the set of failed executions, and that in the whole set of executions (both passed and failed). CBI computes the increase from the latter probability to the former one. The predicates are sorted in descending order of such increases. To be fair in comparing with statement-level techniques (such as *Tarantula*), the predicate-level technique CBI has been adapted to the statement-level by Yu et al. [112]. We use their adapted version SBI [112] in our experiment. The ranking formula is as follows.

$$R_{SBI}(s_i) = failed(s_i)/(failed(s_i) + passed(s_i))$$

We further include two more techniques, *Jaccard* [1] and *Ochiai* [1]. They are similar to *Tarantula* except that they use different ranking formulas and do not use the confidence value $Conf(s_i)$ to resolve tie cases. They have been compared or evaluated in [106][112][120]. The respective ranking formulas for *Jaccard* and *Ochiai* are as follows.

$$R_{Jaccard}(s_i) = failed(s_i)/(m + passed(s_i))$$
$$R_{Ochiai}(s_i) = failed(s_i)/\sqrt{m \times (passed(s_i) + failed(s_i))}$$

These four peer techniques are used in the experiment of [112].

5.4.3 Experimental Setup

Faulty Version Selection Strategy

Each fault in each version of the subject programs is used as one faulty version in our experiment. We view that the four UNIX tool programs have realistic scales, so we further use them to simulate multi-fault versions. Some faults in the same version are conflicting and cannot be enabled simultaneously. To avoid conflicts, we exclude such faults, and use the remaining faults as the fault pool. We randomly generate 10 two-fault versions and 10 three-fault versions based on the fault pool. (Each two-fault version contains two faults, and each three-fault version contains three faults.) The same generation process and same numbers of faulty versions for two-fault and three-fault experiment are also used in [112]. However, we use more realistic program subjects.

In addition, with regard to a faulty version, if many test cases are failure-causing test cases, it means that fault in such a faulty version is very easy to be



revealed. On the other hand, if a faulty version comes with no failure-causing test case, it may be a very hard fault. In our experiment, we exclude both the best cases and the worst cases. In particular, we use the magic number 25 (more than 25% of all test cases are failure-causing ones) as the threshold to determine a best case. Similar strategies are also used in previous work [40][52][68]. (For the worst case, another reason is that our and the other peer techniques require the existence of failed executions.)

Test Suite Generation

For each faulty version, we generate a test suite following the adequacy strategy of statement coverage. We randomly select one pair of passed and failure-causing test cases from the associated test pool, and then randomly add in an unselected test case as long as it increases the cumulative statement coverage of the generated test suite, until the created test suite achieves a statement coverage identical to that of the test pool. A similar strategy is used in [64].

Experiment Scenarios

In our experiment, we iterate two scenarios to evaluate the effectiveness of those techniques. In the first scenario, we use both passed and failure-causing test cases in the generated test suite as input to demonstrate a conventional statistical fault-localization environment. In the second scenario, we pick out all failure-causing test cases from the generated test suite and use them as input to demonstrate a fault-localization environment with only bug report provided. We denote these two scenarios as the “pass and fail” scenario and the “fail only” scenario.

Experimental Environment

Our experiments are carried out on a Dell PowerEdge 2950 server with two Xeon 5430 processors, serving Solaris UNIX with the kernel version of Generic_120012-14. The version of the C++ compiler is Sun C++ 5.8.

We use `gcov-3.4.3` (flags `-a -b -c -f -u`) to collect the execution counts of statements. For every statement, we use all the execution count data provided by `gcv`, without sampling. We also follow [68] to exclude all test cases that produce crashed program executions (that cannot be processed by `gcv`).



5.4.4 Effectiveness Metrics

In our experiment, we examine the list until the faulty statements can be isolated, to measure the performance of a technique. In particular, if a fault lies in some non-executable statement (such as the case of a constant definition fault or a statement omission fault), dynamic information does not help locate it. To reflect the techniques' fault-localization capability on such faults, as previous work [64][120], we mark the directly influenced or the adjacent (to the fault) executable statement as a (pseudo-)faulty statement, and evaluate the effectiveness of these techniques on locating it. We check the statements in the ranked list according to their ranks (statements with same rank are examined as a whole), and calculate the percentage of statements examined when reaching the first faulty statement. We define the metric as follows.

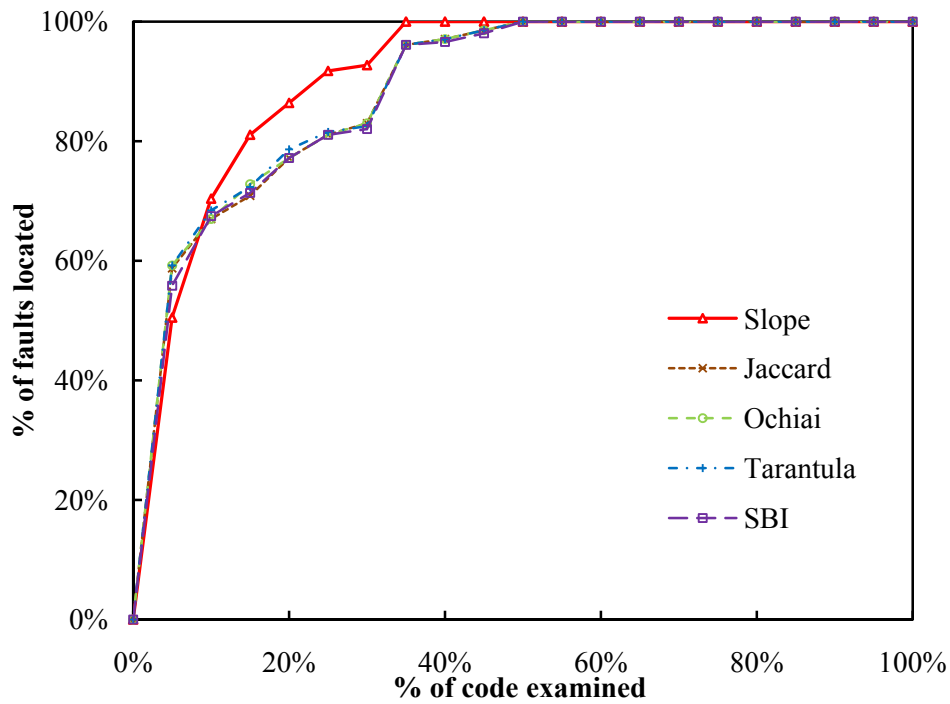
$$\begin{aligned} & \text{\% of code examined} \\ &= \frac{\text{number of examined statements}}{\text{total number of statements}} \times 100\% \end{aligned}$$

5.4.5 Results and Analysis

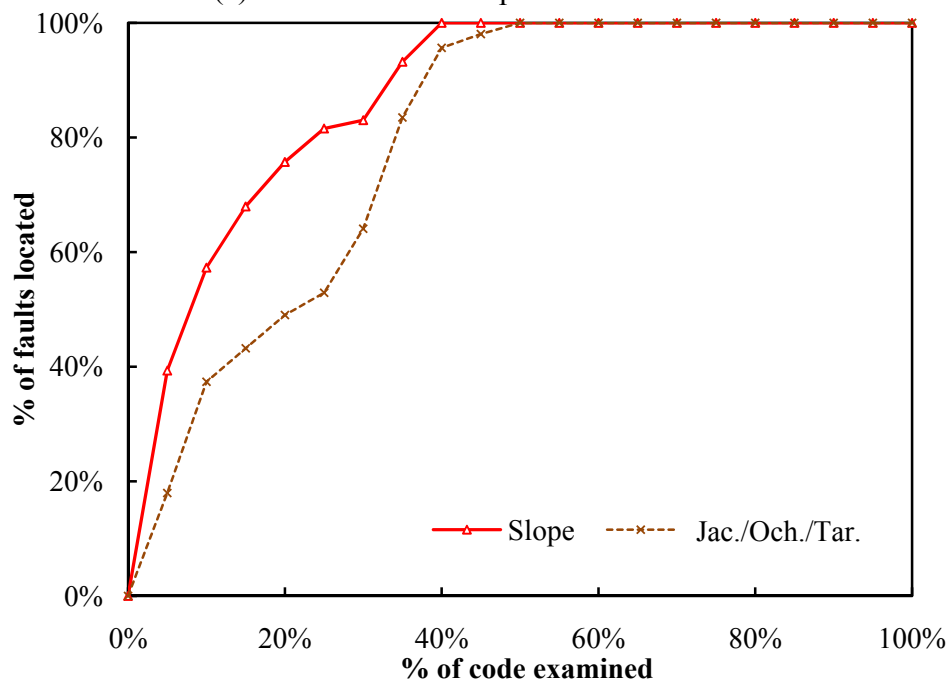
Overall Results on All Programs

Figure 5.4 shows the overall results of those techniques on all 206 faulty versions in the “pass and fail” scenario (shown in plot Figure 5.4(a)) and also in the “fail only” scenario (shown in plot Figure 5.4(b)). In each plot, the X-axis stands for the percentage of code examined, the Y-axis stands for the percentage of faults located within the code examined as indicated by the X-coordinate. Therefore, the curve in each plot shows the trend of speed of a technique locating faults in faulty versions. Our method **Slope** is drawn using the red (heavier) curve with hollow triangles showing the data points sampled at 5%, 10%, 15% code examined points and so on. From the figure, we see that all curves start at the 0% code examining point with 0% faults located, and reach the 100% faults localization before going to the 100% code examining point. It can be explained as following. First, no fault can be located when examining no code. Second, since we include all statements (executable or non-executable, see the metric in Section 5.4.4) when checking and the fault must exist in executable statements (see the mark of faulty statement in Section 5.4.4), all faults can be located without the need of examining all statements.





(a) Overall results in “pass and fail” scenario



(b) Overall results in “fail only” scenario

Figure 5.4: Overall results on all faulty versions

We observe from Figure 5.4(a) that, in the code examined range from 10% to 100%, the curve of **Slope** is above, or at least overlap with, the curves of the peer techniques. For example, when examining up to 20% code, **Slope** can locate faults in 86% (178 out of 206) faulty versions; while **Jaccard**, **Ochiai**, **Tarantula**, and **SBI** can only locate faults in 77% (159 out of 206), 77% (159 out of 206), 79% (162 out of 206), and 77% (159 out of 206) faulty versions, respectively. For the other code examining points, say 30%, the results can be similarly explained. However, when examining up to 5% code, **Slope** is not as effective as the others do and has space of improvements.

One can easily observe that when there is no passed execution, the ranking formulas for **Jaccard**, **Ochiai**, and **Tarantula** degenerate to output same ranked list of statements. Therefore, in Figure 5.4(b), we use a curve annotated with “**Jac./Och./Tar.**” instead of showing three overlapping curves. Additionally, we do not show the effectiveness of **SBI** in this scenario because **SBI**'s formula gives all executed statements the same rank and does not have any fault-localization capability in this scenario.

We observe from Figure 5.4(b) that the curve of **Slope** is always above, or at least overlap, the curves of the other techniques. For example, when examining up to 10% code, **Slope** can locate faults in 57% (118 out of 206) faulty versions; while **Jaccard**, **Ochiai**, and **Tarantula** can only locate faults in 37% (77 out of 206) faulty versions. The other checking points can be similarly explained. In the first half code examining range, 0% to 50%, **Slope** always locates more faults than the others do. After 50% code has been examined, all the faults have been located. Interestingly, we observe that the effectiveness of **Slope** in Figure 5.4(b) is close to that of Figure 5.4(a); whereas the effectiveness of peer techniques shows very noticeable degradation.

Overall, we find that **Slope** has a promising fault-localization capability. It is however less effective than the peer techniques in the “pass and fail” scenario in small code examination range. Moreover, when passed executions are not reliable or unavailable to be used, its effectiveness does not degrade as dramatic as the peer techniques do.

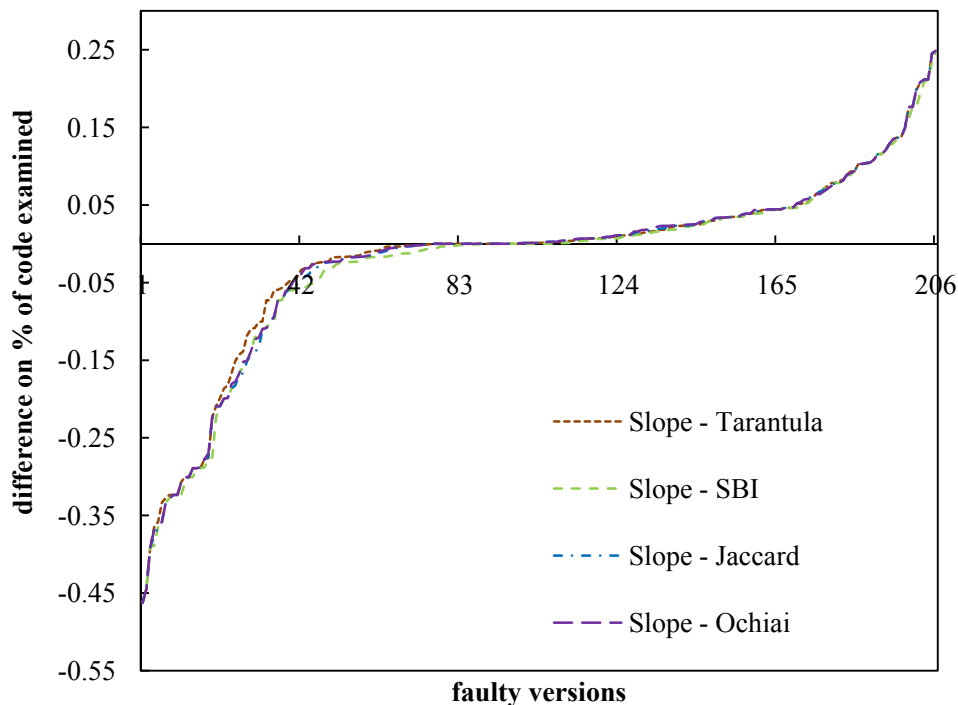
In the next section, we continue to report the statistics of individual results on each faulty version. We further note that we originally plan to show the plots of individual programs; however, there are 11 programs in total and each program has two plots for the two scenarios, we do not condense them into this thesis.

Individual Results on Each Faulty Version

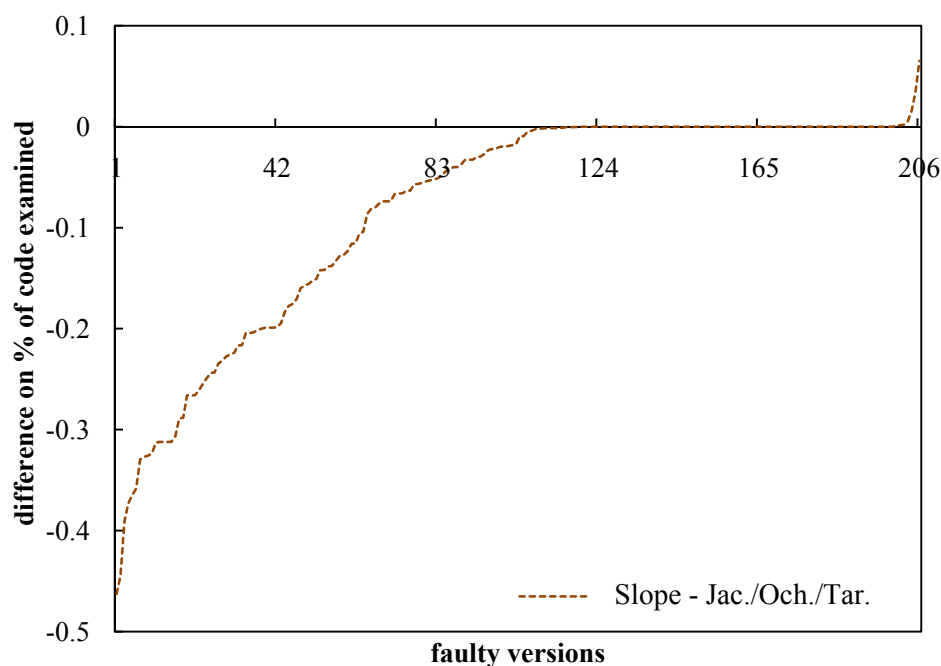
To give a clear representation, we use Figure 5.5 to compare the effectiveness of these techniques on each individual faulty version. The X-axis stands for



the 206 faulty versions (with 186 single-fault versions and 20 multi-fault versions). The Y-axis is the “difference of percentage of code examined” to locate the fault indicated by X-axis. Note that, for the ease of presentation for each curve, the 206 faulty versions are placed in the ascending order of the Y-value of their coordinates. Take the 41st point for the curve indicated as “Slope – Tarantula” for illustration (that is, $x = 41$), **Slope** needs to examine 0.342% of all code to locate it, while **Tarantula** need to examine 4.574% of all code before reaching the fault. Therefore, the difference (in the figure) is shown as $0.342\% - 4.574\% = -4.232\%$. Such a negative value means the effectiveness of **Slope** is better than that of **Tarantula**. As such, each curve in the figure indicates the difference on effectiveness between **Slope** and that of the corresponding technique. Intuitively, the larger the area below X-axis of enclosed portion by that curve and axis, the more effective is **Slope** than that technique.



(a) Effectiveness difference in “pass and fail” scenario



(b) Effectiveness difference in “fail only” scenario

Figure 5.5: Effectiveness difference on all faulty versions

In plot Figure 5.5(a), we observe that, for a majority of X values (that is, for the same fault), the points of Jaccard, Ochiai, Tarantula and SBI are close to one another. It means that, for the majority of the 206 faulty versions, their effectiveness has strong correlations. We also calculate the area below the X-axis and that above the X-axis, and find that the former is always larger than the latter. That helps explain why Slope has an average improvement over the other techniques. In Figure 5.5(b), we have the same observation, but much more noticeable. For about half of the faulty versions, Slope performs better; only for a small number of faulty versions (7 out of 206), can Jaccard, Ochiai, and Tarantula catch up with Slope.

Statistics of Individual Results

Table 5.2: Statistics of individual results on 186 sing-fault versions

	“pass and fail” scenario					“fail only” scenario	
	Slope	Jaccard	Ochiai	Tarantula	SBI	Slope	Jac./Och./Tar.
min	0.007%	0.007%	0.007%	0.007%	0.010%	0.007%	1.130%
max	34.884%	47.231%	47.231%	47.231%	47.231%	36.047%	47.231%
mean	8.933%	10.491%	10.421%	10.660%	10.815%	13.248%	21.776%
stdev	9.754%	13.392%	13.379%	13.494%	13.445%	12.121%	13.498%

Table 5.2 and Table 5.3 show the statistics of effectiveness for all the experimented techniques on the 186 single-fault and 20 multi-fault faulty versions, respectively. For the reason as stated previously, for the “fail only” scenario, we group the effectiveness of Jaccard, Ochiai, and Tarantula into the same column and show them in the “Jac./Och./Tar.” column. The best technique in each row has been highlighted to ease readers to reference.

Let us first focus on the “pass and fail” scenario. In Table 5.2, let us first take the “min” row for illustration. It means that, in the best case, Slope needs 0.007% code examination effort to locate a fault; it is comparable to the effectiveness of Jaccard, Ochiai, Tarantula, and SBI in the best case, which are 0.007%, 0.007%, 0.007%, and 0.010%, respectively. In the worst case (“max” row), Slope needs 34.884% code examination effort to locate a fault; it is better than the peer techniques. When focusing on the “mean” and “stdev” rows, which show the average and standard deviation of code examination effort to locate a fault, Slope is better than the peer techniques.

In the “fail only” scenario, similar observations can be found. The statistics on the “min”, “max”, “mean”, and “stdev” parameters show that Slope is also the most effective among the five.

Let us next move to Table 5.3, which shows the results on the 20 multi-fault faulty versions. From the worse case line (“max” row), we observe that Slope locates all faults when examining up to 13.751% code; while Jaccard, Ochiai, Tarantula, and SBI need to examine much more code (requiring to

Table 5.3: Statistics of individual results on 20 multi-fault versions

	“pass and fail” scenario					“fail only” scenario	
	Slope	Jaccard	Ochiai	Tarantula	SBI	Slope	Jac./Och./Tar.
min	0.008%	0.079%	0.079%	0.062%	0.606%	0.007%	1.078%
max	13.751%	38.040%	37.184%	37.500%	45.725%	13.678%	14.392%
mean	3.564%	7.764%	7.510%	3.723%	8.529%	4.004%	5.835%
stdev	3.746%	11.908%	11.634%	8.235%	11.692%	3.694%	3.476%



Table 5.4: Results of Pearson correlation test to compare the effectiveness of techniques

	“pass and fail” scenario				“fail only” scenario
	$\phi = \text{Jaccard}$	Ochiai	Tarantula	SBI	Jac./Och./Tar.
Slope w.r.t. ϕ	0.442	0.444	0.465	0.443	0.614
SBI w.r.t. ϕ	0.970	0.970	0.983		
Tarantula w.r.t. ϕ	0.962	0.962			
Jaccard w.r.t. ϕ	1.000				

examine around 40% of code) to locate the last fault. We further observe that, in the best case (the “min” row), Slope needs to examine only 0.007% code to locate a fault, while Jaccard, Ochiai, Tarantula, and SBI need to examine 0.079%, 0.079%, 0.062%, and 0.606% code, respectively, to locate a fault. For the “fail only” scenario, the result of Slope is also interesting. The table shows that our technique can, on average, locate faults by examining 4% of code; whereas the peer techniques requires to examine 5.8%, or equivalently, using our technique, on average, developers can examine around 100 less statements to locate the faults.

Given that a program having multiple faults is more common than a program having single fault, we find that the result of Slope indicates that, statistically, Slope can be more realistically applied to debug real-life programs than the peer techniques.

Hypothesis Testing to Compare the Techniques

In previous sections, we have the observation that, in terms of effectiveness, our technique Slope, on average, is better than the peer techniques. To know whether the difference is statistically significant enough, we conduct the Student’s t-test [125] to compare the effectiveness of Slope with those of Jaccard, Ochiai, Tarantula, and SBI. In each of the two scenarios, we test the null hypothesis on “*the effectiveness of Slope and another technique Φ , on the 206 versions, come from same distribution*”, where Φ is replaced by one of Jaccard, Ochiai, Tarantula, and SBI. The results are shown in Table 5.5. Let us take the first cell for illustration. The cell, means “Slope w.r.t. Jaccard” in the “pass and fail” scenario, and has a result of 0.033. It means that by 3.3% of chance, the values of effectiveness for Slope and Jaccard come from same distribution. Therefore, we may reject the null hypothesis at 5% significance level. The table shows that all five similar null hypotheses can be rejected at the 10% significance level and four of them can even be rejected at the 5% significance level. In summarize, at the aforementioned significance levels and



Table 5.5: Results of t-test hypothesis testing to compare the effectiveness of techniques

	“pass and fail” scenario				“fail only” scenario
	$\phi = \text{Jaccard}$	Ochiai	Tarantula	SBI	Jac./Och./Tar.
Slope w.r.t. ϕ	0.033	0.041	0.062	0.012	< 0.001

the mean effectiveness of **Slope** is better than the peer techniques, we conclude that **Slope** is better than the other four techniques.

Case Analysis

Through case-by-case observation, we find that for 61% (125 out of 206) faulty versions, the effectiveness difference between **Slope** and each of the other techniques exceeds 5% code examination effort. On the other hand, for only 11% (23 out of 206) faulty versions, the effectiveness difference among **Jaccard**, **Ochiai**, **Tarantula**, and **SBI** can exceed 5%. Therefore, we want to know whether **Slope** and the other techniques perform closely simply because they just locate the same fault in a similar extent (in terms of code examination effort). To do that, we perform the Pearson’s correlation test [125] on the effectiveness (on the 206 faulty versions) of pairs of these techniques. The results of correlation test are shown in Table 5.4. We observe that the correlation of effectiveness for **Jaccard**, **Ochiai**, **Tarantula**, and **SBI** are always above or equal to 0.962. That means their effectiveness have strong correlations among one another. However, the correlations of **Slope** with them are always less than or equal to 0.614, which is not a strong correlation. Judging from the results, we deem that **Slope** works very differently from the peer techniques to locate faults successfully.

For instance, we also use one case to illustrate their differences. In faulty version v28 of program **tcas** (Figure 5.6), the fault lies on a **return** statement so that the execution of any test case exercises it. Such a fault is very common in programming [47]. Take **Tarantula** as illustration, such a faulty statement has identical fault-suspiciousness score with most statements in the main module. They are indistinguishable to either the formula or the tie breaking strategy of **Tarantula**. **Tarantula** ranks it as the 56th element in the resultant list and examines 32.370% code to locate it. On the other hand, the faulty statement has larger execution count in the execution of failed executions than in those of passed executions. Our technique **Slope** accurately gives high fault-suspiciousness score to this faulty statement rather than the other statements, and ranks it as the 6th element in the resultant statement list. Finally, **Slope** examines 4.942% code and locates this fault.

```

L63 return ((Climb_Inhibit == 0) ?
           Up_Separation + NOZCROSS : Up_Separation);

/* wrong return expression */
// return (Climb_Inhibit ?
//         Up_Separation + NOZCROSS : Up_Separation);

```

Figure 5.6: Faulty version v28 of tcas

Impact Factors on Effectiveness

We observe that **Slope** has better effectiveness in the “pass and fail” scenario than in the “fail only” scenario. It is understandable because more information (come from the passed executions) is provided in the former scenario. However, there are some drawbacks. For example, the standard deviation in the “pass and fail” scenario is larger than that in “fail only” scenario. One factor could be the presence of coincidental correctness [106] related to passed executions.

Moreover, to apply the effectiveness metric on the multi-fault programs, it appears to locate the first fault earlier, than on the single-fault programs. This could be due to that the chance of hitting faulty statement just increases. Thus, the detailed analysis on the effectiveness on single-fault programs and multi-fault programs should be handled separately.

5.4.6 Threats to Validity

Internal Validity

We have checked the correctness of our tool carefully and removed all bugs found. We choose a Dell server running on Solaris UNIX operating system and use the Sun Studio C++ compiler. To make the results reliable, we have carefully checked our platform.

Construct Validity

We include four representative, existing and non-author techniques in our experiment. However, there exist other fault-localization techniques. For example, **CT** [36] uses pairs of passed and failure-causing test cases to identify faults; **SOBER** [76][77] is a predicate-level fault-localization work. Since the peer techniques used in our experiment are at statement-level and make use of whole test suite, it appears not fair to directly compare **SOBER** or **CT** with them.



We use 11 C programs to verify the effectiveness of our model. They are equipped with test pools. We follow [112] to create multi-fault versions and generate test suites. However, the use of other subject programs, test suite generation strategy, or multi-fault version generation strategy may produce different results. Some previous work (such as [76]) ever reported that, the larger the test suite, the better will be the effectiveness of their techniques.

External Validity

Using other metrics may produce different results. In previous work (such as [90]), t-score is used to evaluate the effectiveness of fault-localization techniques. However, there exist reported limitations of the use of t-score (see [18], for example). In addition, t-score may be not suitable for evaluating statement-level techniques. Therefore, we adopt a metric suggested by some later work (e.g., [64][68][69][112]).

Some previous studies tend to focus on the most effective part when evaluating a technique (e.g., the most favorable faulty versions) and ignore the most ineffective part. In this thesis, we adopt to use the statistics parameters (e.g., mean) to evaluate them. We believe that it is a more fair account on evaluating the effectiveness of fault-localization techniques. Yu et al. also use the mean metric to compare techniques [112].

5.5 Summary

In this chapter, we have proposed a new model and developed a technique Slope with two formulas, which share the same idea (model) and can be applied to scenarios with or without passed executions, respectively. Our underlying model first collects the execution counts of statements and, for each statement, calculates the fraction of failed executions with respect to all executions having the same execution count. It then calculates the failing rate accordingly. Considering each tuple of \langle failing rate, execution count \rangle as a point in two-dimensional space, the model lines up these points and uses the slope of the line as the mean of the signal of suspiciousness and the fitting error as the noise to the signal. This chapter has developed a formula based on the idea of signal-to-noise ratio. Our empirical study has shown that Slope produced promising results. After that, we continue to eliminate the dependency on passed executions by further approximation, which make our technique (Slope) effectively work in scenarios where passed executions have not been available or unreliable to use.

The major contribution of this chapter is twofold. (i) A novel and effective dynamic fault-localization technique that can work in absence of passed executions is developed. (ii) We demonstrate a “fail only” scenario and



present the first empirical evaluation results of this kind of technique. The results show that this technique is empirically promising.

Chapter 6

Non-parametric Hypothesis Testing Method used in Predicate-based Statistical Fault-localization Techniques

In the previous chapter, we perform a finer-grained investigation based on previous statistical predicate-level fault-localization techniques. However, we find that all these predicate-level fault-localization techniques make an assumption that the execution spectra of predicates form specific distributions. Since it is common that applying a model on an unmatched distribution will generate inaccurate result, we, in this chapter, investigate the use of different non-parametric and parametric methods in the predicate-level fault-localization techniques.

We first recall the nature of predicate-level fault-localization techniques, and then use a motivating example to demonstrate the need of non-parametric hypothesis testing methods. After that, we raise research questions to study the effectiveness and validity of using different non-parametric and parametric hypothesis testing methods in predicate-level statistical fault localization, and then use empirical study to answer the proposed research questions.

This chapter is partly based on our work in [119]. In this thesis, further extension is made on the basis of that paper.

6.1 Background

Predicate-level statistical fault-localization techniques presume that, for predicates near the fault position, their evaluation results (successes or failures) are



highly correlated to the successes or failures of the program executions. Hence, identifying effective program predicates and formulating correct and robust statistic comparisons are important for such techniques.

Since these previous predicate-based techniques propose various parametric hypothesis testing models to describe the feature spectra, to apply their self-proposed hypothesis testing models, they set up underlying presumptions that the program feature spectra form specific kinds of known distributions. Let us take the two representative existing predicate-based fault-localization techniques CBI [74][75] and SOBER [76][77] for illustration. Both of these two techniques estimate how much each individual predicate is related to fault and accordingly rank all the predicates to generate a suspicious list, to guide searching fault locations in faulty program. Note that a predicate may be evaluated to be *true* or *false*, or even not evaluated, in a program execution. CBI [74][75] checks the probability of that predicate to be evaluated *true* in all the failed executions and that probability in all the executions (irrespectively of whether passed or failed). It then measures the *increase* from the former to the latter, and uses such increase to indicate how much that predicate is related to fault(s). SOBER [76][77] does not only record whether or not a predicate is evaluated to be *true* or *false*, but also the number of times that predicate is evaluated to be *true* or *false*, with respect to a program execution. It defines *evaluation bias* to estimate the chance that a predicate is evaluated to be *true*, with respect to a program execution. For example, let us suppose P is a predicate and $\pi(P)$ is the probability that P is evaluated to be *true*, with respect to a program execution. From the input test suite, we use $\frac{n_t}{n_t+n_f}$ to estimate $\pi(P)$, where n_t and n_f are the numbers of times that P is evaluated to be *true* and that number of times for *false*, respectively. SOBER then makes use of Central Limit Theorem, and constructs a hypothesis test on the existence of difference between the distributions of evaluation biases of $\pi(P)$ for passed executions and failed executions, using the mean and standard deviation statistical parameters. It deems that the larger the difference, the more will P be related to a fault. The aforementioned probability in CBI can be regarded as the mean value of coverage status if we count one for a predicate being evaluated to be *true* or *false*, with respect to a program execution, and count zero for a predicate not evaluated, with respect to a program execution. Therefore, we know that the technique of CBI belongs to a parametric fault-localization technique. The result is that CBI does not distinguish the number of times that a particular program element (a statement or a predicate) has been executed in an execution, and Liu et al. [77] empirically show that such a method can be less accurate than one in which the distributions of evaluation biases in passed executions and failed executions are considered. On the other hand, since SOBER uses the mean



and standard deviation statistics parameters to test the hypothesis constructed, it uses parametric hypothesis testing method. By using the Central Limit Theorem, SOBER test their desired hypothesis with the mean and standard statistical parameters, which are generally used to describe a normal distribution.

However, our empirical study in Section 6.2 on the Siemens suite [40] shows that the assumption of predicate spectra forming normal distribution is not well supported by the empirical data. What is more, the spectra of most predicates in Siemens programs vary a bit from one another and are far from having any known distribution. Hence, a parametric or ad hoc hypothesis testing approach based on the presumption of normal distributed feature spectra may lose its discrimination capability significantly and produce non-robust results. In previous work [62][60], these motivate us to adopt a generic mathematical model in predicate-based fault-localization approach. Based on previous predicate-based fault-localization techniques, we proposed a hypothesis testing framework for fault localization, and adopted a standard non-parametric hypothesis testing method, the Mann-Whitney test, instead of previous self-proposed hypothesis testing models, to use in this model. We conducted an empirical experiment to show that the effectiveness of previous techniques on the Siemens programs is observably improved. However, what is not clear is that either the improvement is due to the use of a non-parametric hypothesis testing method instead of a parametric hypothesis testing method, because non-parametric hypothesis testing method better describe the feature spectra distribution rather than any parametric hypothesis testing methods, or it is due to the use of a standard hypothesis testing method instead of a self-proposed hypothesis testing method, because standard hypothesis testing method are mathematically more robust and elegant?

At the mean time, can we have more confidence on a more general result that a non-parametric hypothesis testing model is in most cases better than a self-proposed hypothesis testing model or a standard parametric hypothesis testing model, when using in a fault-localization environment with feature spectra do not form any known distribution?

In view of the above-mentioned initial study, in this chapter, we further ask a couple of questions: Can the feature spectra of program elements be safely considered as normal distributions so that parametric fault-localization techniques can be soundly and powerfully applied? Alternatively, to what extent can such program spectra be regarded as normal distributions? If the answers to these questions are negative, we further ask the following question: Can the effectiveness of non-parametric fault-localization techniques be really decoupled from the distribution shape of the program spectra?



6.2 Motivation

In this section, we use a code excerpted from the faulty version “v1” of program “tot_info” from the Siemens suite [40] to motivate our work. Figure 6.1 shows the code excerpted, where seven predicates are included, labeled as P_1 to P_7 . Close to predicate P_4 , there lies in a statement omission fault. According to the location of the fault, predicate P_5 is the directly affected predicate. The aim of predicate-based fault-localization techniques is to find such predicates P_4 or P_5 that are mostly close to fault, in terms of their positions (i.e., line number) in the code. Note that, according to previous studies [47], statement omission fault is hard to identify, even if the execution of a failure-causing test case is traced step-by-step. Previous studies [62][74][75][76][77][119][122][123] tell us the feature spectra of predicates can be used as good indicator of fault location. Therefore, for each of the predicates P_1 to P_7 , we use two histograms [62] to show their distributions of evaluation biases in passed executions and failed executions, respectively, in Figure 6.2 and 6.3. Take the P_7 plot for passed executions in 6.3 as illustration. The left-most bar means that, for 82 passed executions, the evaluation biases of P_7 are in the range of [0.65, 0.66).

We have the following observations on these histograms: For each predicate among P_1 , P_2 , P_3 , P_6 , and P_7 , the evaluation biases captured from passed execution and those from failed executions form similar histograms. For predicate P_4 or P_5 , the histogram formed by evaluation biases from passed executions and that formed by evaluation biases from failed executions have observable difference between each other. It consolidate the underlying heuristic in previous studies that the distribution differences of evaluation biases over passed and failed executions can be good indicators of the fault relevance of predicates.

However, the model used to describe the feature spectra and test their similarity should be determined with the distribution of feature spectra considered in. We notice that the histograms in Figures 6.2 vary very much, and no one of them can be quantified to have a known (e.g., Gaussian or normal) distribution. In our previous studies, we report that “for nearly 60% of a total of 10042 predicates, the assumption of their evaluation biases forming Gaussian distribution is rejected at the 5% significance level [62]”. As a result, we deem that, as far as the programs under study can represent, assuming the evaluation biases of predicates to have normal distribution is unrealistic. Take the histograms of P_5 as an example. It is obviously far from a normal distribution. If we deem its evaluation bias to have normal distribution and calculate the statistical parameter **mean**, such a value in left plot (passed executions) and that in right plot (failed executions) are 0.101 and 0.325, respectively. Though we may intuitively draw



Program : tot_info.c [62]	
<i>P</i> ₁ :	if (rdf ≤ 0 cdf ≤ 0) { info = -3.0; goto ret3; }
	⋮
<i>P</i> ₂ :	for (i = 0; i < r; ++i) { double sum = 0.0;
<i>P</i> ₃ :	for (j = 0; j < c; ++j) {
<i>P</i> ₄ :	long k = x(i,j);
<i>E</i> ₁ :	if (k < 0L) { info = -2.0; /*goto ret1;*/
	} sum += (double)k;
	} N += xi[i] = sum;
	}
<i>P</i> ₅ :	if (N ≤ 0.0) { info = -1.0; goto ret1;
	}
<i>P</i> ₆ :	for (j = 0; j < c; ++j) { double sum = 0.0;
<i>P</i> ₇ :	for (i = 0; i < r; ++i) sum += (double)x(i,j); xj[j] = sum;
	}
	⋮
	ret1:

Figure 6.1: Excerpt from faulty version “v1” of program “tot_info”

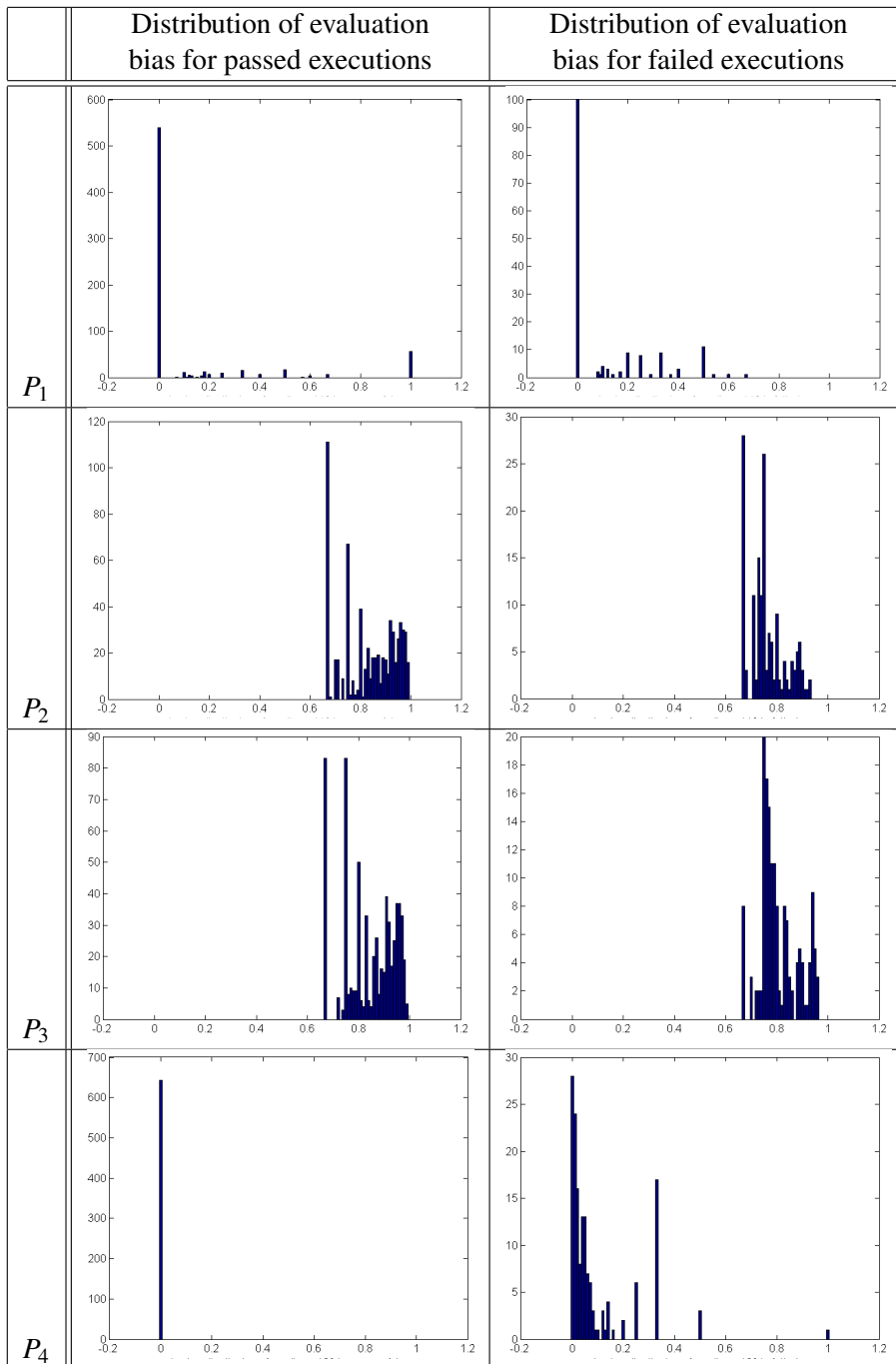
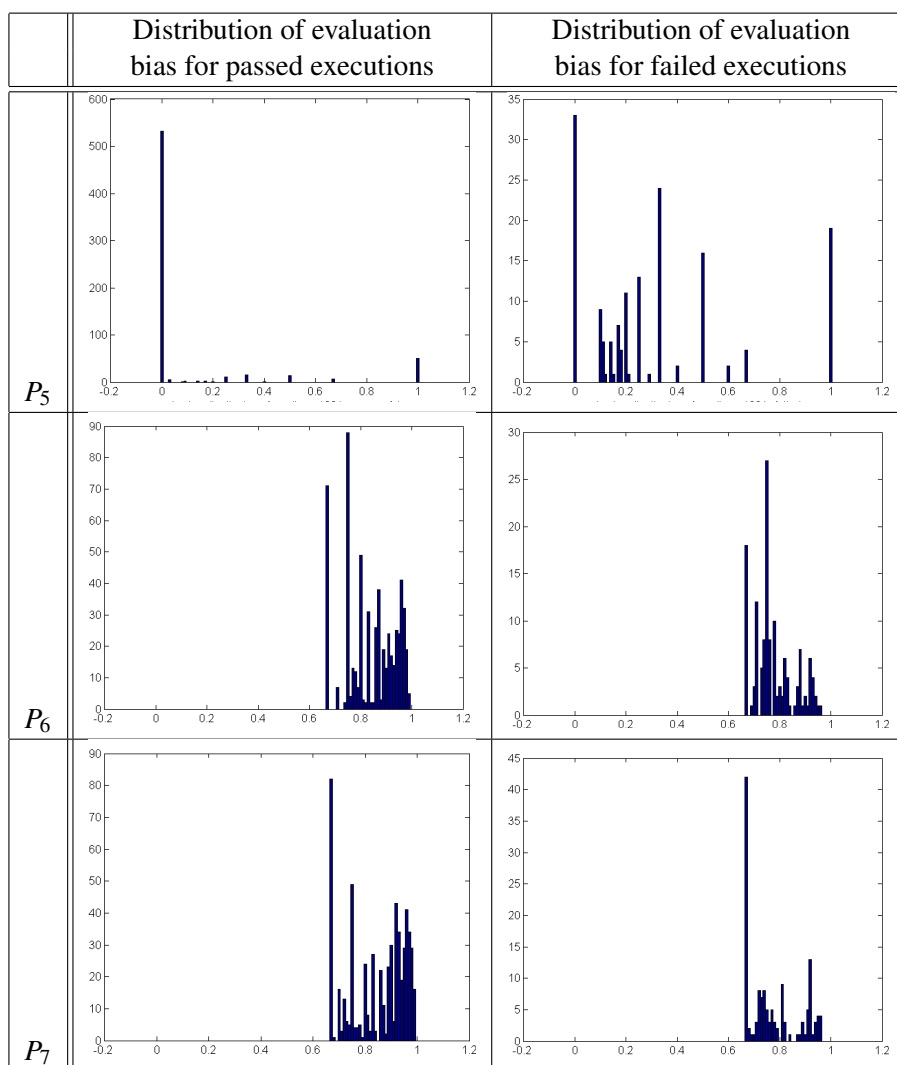


Figure 6.2: Distributions of evaluation biases for predicates P_1 to P_4



Figure 6.3: Distributions of evaluation biases for predicates P_5 to P_7

a conclusion that these two numbers are not equal and there must be huge difference between them, there is no scientific support to quantify such a difference. Besides, none of the histograms in Figures 6.2 and 6.3 resembles a normal distribution. For each predicate of every program in the Siemens suite, we have conducted the standard Jarque-Bera test to determine whether its evaluation bias follows a normal distribution. The results, which are given in Section 6.5.5, show that, as far as the programs under study can represent, it is unrealistic to assume normal distributions for the evaluation biases of predicates. On the other hand, if we use non-parametric hypothesis testing method to compare these two distributions, their differences are scientifically measured. This is because the concept of non-parametric hypothesis testing methods does not rely on any presumption of distributions. To rank the predicates in their order of suspiciousness to be fault, we only need to know the relative order of each two predicates in terms of the differences between their “passed” histogram and their “failed” histogram. Therefore, we may conduct non-parametric hypothesis testing on them and use the p-value result of such test as a measurement of suspiciousness to compare two predicates. This is because the less the p-value result is, the more probably the two histogram are from same population, and accordingly, the more difference between them.

From the above observations, the assumption that the evaluation biases of predicates form normal distributions is not well-supported by the empirical data. Furthermore, only a small number of test cases can reveal failures in practice, and the number of successful test cases is not large either. Because of that, in our previous work [62], we proposed a non-parametric hypothesis testing model, and advocated the use of a non-parametric predicate-based fault-localization technique. However, the applicability of non-parametric fault-localization technique has not been fully investigated. Can the feature spectra of program entities be safely considered as normal distributions so that parametric techniques can be applied rigorously? This motivates the study in this thesis.

6.3 Our Fault-localization Framework

In this section, we explore a model for ranking fault-relevant predicates to facilitate locating faults in programs.

6.3.1 Preliminaries

We first revisit the notion of program predicates and evaluation biases [74][75][76][77].



Liblit et al. [74][75] list out three types of program locations, with which a set of predicates are associated, to sample the execution spectra, with respect to each passed execution and each failed execution. Such program locations and associated predicates include:

1. *Branches*: At each branch statement, for example, an “if” statement, a “while” statement, CBI tracks the conditional *true* and *false* branches via a pair of program predicates, which monitor whether the corresponding branches have been taken. SOBER further collects the number of times that the branches have been taken in an execution.
2. *Returns*: At each return statement (of a function module), six predicates are tracked to find whether the returned value r satisfies $r < 0$, $r \leq 0$, $r > 0$, $r \geq 0$, $r = 0$, and $r \neq 0$, respectively. Both CBI and SOBER collect evaluation biases for these predicates.
3. *Scalar-pairs*: To track the relationship between a variable and another variable or constant in each assignment statement, six predicates (similar to those for return statements above) are adopted by CBI. For example, six predicates are tracked to find the Boolean relationship of $x > y$, $x \geq y$, $x < y$, $x \leq y$, $x = y$, and $x \neq y$, for an assignment statement $x := y$. On the other hand, SOBER experimentally verifies and concludes that *not* tracking these predicates will not degrade the fault-localization quality when using the Siemens suite.

A program predicate P may thus be evaluated multiple times in an execution. Each program predicate may be executed more than once in an execution. Each evaluation will give either a *true* or a *false* value. We thus give the notion of evaluation bias (see Definition 4.2.1 in Chapter 4) to estimate the probability of a predicate being evaluated as true in an execution.

6.3.2 Problem Settings

We first use $\{P_1, P_2, \dots, P_m\}$ to denote the set of predicates in a faulty program. We further use R and R' to denote the set of passed executions and the set of failed executions. To differentiate the evaluation bias of a predicate in passed executions from those in failed executions, we use $E_{i,j}$ to denote the evaluation bias of predicate p_i in a passed execution $r_j \in R$. Similarly, we use $E'_{i,k}$ to denote the evaluation bias of predicate p_i in a failed execution $r'_k \in R'$. To facilitate locating the fault, our aim is to generate a predicate list, which arranges predicates P_1, P_2, \dots, P_m in descending order of how much each of them is related to fault.



6.3.3 Our Framework

Following our previous work [62][60], we use,

$$R(P_i) \approx \text{Diff}(\{E_{i,1}, E_{i,2}, \dots, E_{i,|R|}\}, \{E'_{i,1}, E'_{i,2}, \dots, E'_{i,|R'|}\}) \quad (6.1)$$

to measure the difference between the two sample sets.

As suggested by the rational of hypothesis testing methods, the p-value result of a hypothesis testing method is the probability of getting a result as extreme as observed, presuming the null-hypothesis. We construct the null-hypothesis as “the two sets come from same kind of population”, and use the p-value of a hypothesis testing method to replace the ranking function in Equation (6.1). For the program feature spectra in failed executions and passed executions, there is no scientific supports for the mapping of similarity of their distributions and the magnitude of the p-value of hypothesis testing on their distributions. However, we know that the less the p-value is, the less they are from same kind of population and the more difference there exist between them. Since we only need to compare those predicates and know their relative order of suspiciousness, we sort all the predicates according to the corresponding ranking scores computed. Such a resultant suspicious predicate list is helpful for programmers to locate fault in programs [74–77, 90].

6.4 Research Questions

To measure the difference between the two sample sets, a previous promising way is to use a parametric hypothesis testing method. However, according to standard statistics textbooks such as [78], a parametric hypothesis testing can be meaningfully applied only if

- C1: The two sample sets are independently and randomly drawn from the source population;
- C2: The scales of measurement for both sample sets have the properties of an equal interval scale;
- C3: The source population(s) can reasonably be assumed to have a known distribution.

In cases where the data from two independent samples fail to meet any of these requirements, it is a well-known advice to use a non-parametric alternative. This is further supported by our empirical study presented in Section 6.2, which



Table 6.1: Techniques we are interested in

Technique Class	Explanation Explanation	Parametric or Non-Parametric?	Standard or Self-proposed?
TC1	aforementioned previous existing techniques, such as CBI and SOBER	parametric	self-proposed
TC2	using parametric hypothesis testing methods in our framework	parametric	standard
TC3	using non-parametric hypothesis testing methods in our framework, such as using Mann-Whitney test	non-parametric	standard

shows that the underlying data populations are indeed far from a known distribution model. The robustness of non-parametric hypothesis testing also frees us from having artificial configuration parameters.

The Mann-Whitney test is a widely used non-parametric test method. It can be used to compare medians of two non-normal distributions. In our previous work [62], we use the Mann-Whitney test to conduct hypothesis testing. The P-score result of our method using the Mann-Whitney test (Mann-Whitney for short) is the p-value of the Mann-Whitney test, which measures the difference between the evaluation bias for all passed executions and those for all failed executions in the Siemens suite, for a given predicate. Such results reflect how much a predicate is fault-relevant. We use such p-values to sort the predicates, and generate a list of predicates. In such a predicate list, the predicates are sorted in descending order of how much each of them is fault-relevant. Similar procedure is also explained in Section 6.3.2. We then apply the metrics in Equation (6.2) to evaluate the effectiveness of fault-localization techniques. The empirical results [62][119] showed that using the Mann-Whitney test is effective. However, does it imply that a non-parametric hypothesis testing method is more proper than an ad hoc self-proposed hypothesis testing method in predicate-level fault localization? To answer this question and enhance the conclusion in our previous work [62][119], we classify the techniques under investigating into three classes (as shown in Table 6.1) and design the following research questions.

Q1: Comparing to TC1 techniques, are TC2 techniques more effective?

Q2: Comparing to TC2 techniques, are TC3 techniques more effective?



Q3: Comparing to TC1 techniques, are either TC2 or TC3 techniques more effective?

The research question *Q1* involves both TC1 techniques and TC2 techniques to help make clear whether standard hypothesis testing methods perform better than self-proposed hypothesis testing methods. The research question *Q2* involves TC2 techniques and TC3 techniques to help make clear whether non-parametric hypothesis testing methods perform better than parametric hypothesis testing methods. The research question *Q3* enhances to answer whether standard hypothesis testing methods, irrelevant to parametric or non-parametric, always perform better than self-proposed hypothesis testing methods.

Further, we have another two questions to help understand some important properties of our techniques. Research question *Q4* investigates the scalability issue of our framework; research question *Q5* investigates the efficiency issue of our framework.

Q4: With the increasing of test suite size, do T2 and T3 techniques gain more effective fault-localization results?

Q5: Do T1, T2, and T3 techniques have comparably practical efficiencies, say running time?

To investigate the applicability of using non-parametric hypothesis testing model for fault localization, we further design the following research questions:

Q6: Is normal distribution common in program spectra (and evaluation biases of predicates in particular)?

The answer to this question relates to whether it is suitable to use parametric hypothesis testing methods on the evaluation biases of predicates for fault localization. If it is not common for the evaluation biases of predicates be normally distributed, the assumption that the program spectra on predicates can be regarded as normal distributions cannot be well supported. It appears not rigorous enough to use parametric hypothesis testing methods on the evaluation biases of predicate for fault localization.

Q7: Is normal distribution common in the program spectra of the most fault-relevant predicates (and evaluation biases in particular)?

Many fault-localization techniques (such as [62][68][70][74][75][76][77]) generate a predicate list, which arranges all the predicates in descending order of their fault relevance. For these techniques, the most fault-relevant



predicates play an important role, since the effectiveness of each technique is mainly decided by the efficiency in locating such predicates in the given predicate lists. Therefore, we also investigate the normality of the most fault-relevant predicates. If the answer to question $Q1$ is *no*, and yet the answer to this question is *yes*, the use of parametric hypothesis testing methods in fault localization may be still acceptable.

$Q8$: Does the normality of evaluation biases of the most fault-relevant predicates correlate with the effectiveness of a non-parametric fault-localization technique?

If the answers to both questions $Q1$ and $Q2$ are *no*, it appears unsuitable to uphold the assumption that the underlying program spectra form normal distributions. It also indicates that the use of a non-parametric fault-localization technique such as the one proposed in our previous work [62], is a viable choice. As such, we further investigate whether the normality of evaluation biases of the most fault-relevant predicates correlates with the effectiveness of a non-parametric fault-localization technique.

6.5 Experimental Evaluation

In this section, we present the experiment to answer research questions. We first introduce the subject programs, the selected effectiveness evaluation metric, and the setup of the experiment. We then report the effectiveness of our fault-localization model, using different hypothesis testing methods, and the effectiveness of CBI and SOBER, over the Siemens suite programs. After that, we investigate the effect of different test suite size on these fault-localization techniques and report their timing issue. Finally, we discuss the threats to validity of experiment.

6.5.1 Subject Programs

In this section, we use the seven Siemens programs, “tcas”, “tot_info”, “replace”, “print_tokens”, “print_tokens2”, “schedule”, and “schedule2”, to validate our idea. Each of the programs has 7 to 41 faulty versions, each of which is seeded with one fault. We show the number of faulty versions for them, the executable line of code, number of test cases in the test pool, and the percentage of failure-causing test cases among all test cases in Table 6.2. All these programs are downloaded from the Software-artifact Infrastructure Repository (SIR [40]).



Table 6.2: Statistics of Siemens suite

Siemens Programs	No. of Executable Faulty Versions	No. of LOC	No. of Test Cases	Percentage of Failed Test Cases
print tokens & print tokens2	17	341–354	4115 - 4130	1.7% - 5.4%
minimum failure rate = 0.001 print_tokens v1 <i>/* Wrong branching around statements */</i> <i>/* case 16 : ch=get_char(...); case 25 : case 32 : token_ptr->token_id=special(next_st); */</i> 224: case 16 : case 32 : ch=get_char(...); case 25 : token_ptr->token_id=special(next_st);				
maximum failure rate = 0.125 print_tokens2 v6 <i>/* Wrong logic or relational operands */</i> 358: if(isdigit(*(str+i+1))) <i>/* i+1 should be i */</i>				
median failure rate = 0.042 print_tokens2 v10 <i>/* Wrong logic or relational operands */</i> 380: { while (<i>str /* str should be str+i */</i>)!=^0)				
replace	32	508–515	5542	2.0%
minimum failure rate = 0.0001 replace v15 <i>/* Wrong logic or relational operands */</i> 241: result = i + 1; <i>/* i+1 should be i */</i>				
maximum failure rate = 0.035 replace v19 <i>/* Missing assignment */</i> 514: <i>/* result = */</i> getline(line, MAXSTR, &result);				
median failure rate = 0.006 replace v14 <i>/* Missing OR-term/AND-term */</i> 370: if(<i>(lin[*i] != NEWLINE) /* && (!locate(lin[*i], pat, j+1)) */</i>)				
schedule & schedule2	19	261–294	2650 - 2710	2.4% - 3.2%
minimum failure rate = 0.001 schedule2 v5 <i>/* Missing the whole if statement */</i> 111: <i>/* if(prio < 1) return(BADPRIO); */</i>				
maximum failure rate = 0.116 schedule v7 <i>/* Missing the whole if statement */</i> 210: <i>/* if(ratio == 1.0) n--; */</i>				
median failure rate = 0.011 schedule v4 <i>/* Wrong logic or relational operands */</i> 207: if(count > 1) <i>/* 1 should be 0 */</i> {				
tcas	41	133–137	1608	2.4%
minimum failure rate = 0.001 tcas v12 <i>/* Wrong logic or relational operators */</i> 118: enabled = High_Confidence <i>/* should be && */</i> (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);				
maximum failure rate = 0.182 tcas v27 <i>/* Missing OR-term/AND-term */</i> 118: enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) <i>/* && (Cur_Vertical_Sep > MAXALTDIFF) */</i> ;				
median failure rate = 0.021 tcas v10 <i>/* Wrong logic or relational operators */</i> 105: return (Own_Tracked_Alt <= <i>/* <= should be < */</i> Other_Tracked_Alt);				
tot_info	23	272–274	1052	5.6%
minimum failure rate = 0.001 tot_info v23 <i>/* Wrong logic or relational operators */</i> 215: for (n = 0 <i>/* 0 should be 1 */</i> ; n <= ITMAX; ++n)				
maximum failure rate = 0.087 tot_info v7 <i>/* Wrong logic or relational operators */</i> 378: if(pi >= <i>/* >= should be > */</i> 0.0)				
median failure rate = 0.017 tot_info v2 <i>/* Wrong logic or relational operators */</i> 85: if(scanf("%ld", &x(i,j)) == <i>/* == should be != */</i> 0)				

Table 6.3: Important fault types for C programs

Orthogonal Detect Classification Class	Fault type [47]
Assignment (22.8%)	A1: Missing assignment (43%) A2: Wrong/extraneous assignment (37%) A3: Wrong assigned variable (11%) A4: Wrong data types or conversion (7%)
Check (26.6%)	C1: Missing OR-term/AND-term (47%) C2: Wrong logic or relational operators (32%) C3: Missing branching around statements (20%)
Interface (7.8%)	I1: Wrong actual parameter expression (63%) I2: Missing return (18%) I3: Wrong return expression (14%)
Algorithm (42.7%)	G1: Missing the whole “if” statement (40%) G2: Missing function call (26%) G3: Wrong function call (8%)

Table 6.2 describes the statistics of subject program. Such statistical information include number of faulty versions for each program, number of executable statements for the faulty versions of each program, number of test cases, and percentage of failure-causing test cases. Also, for the available faulty versions of each program, the minimum, maximum, median failing rate [34], of those faulty versions is reported, with their fault type and code excerpts. Take the program *tcas* as an example. It has 41 faulty versions, each of which has 133 to 137 lines of executable code. There are 1608 test cases available for this program; while 2.4% of them are failure-causing test cases. Among the 41 faulty versions, fault in version *v12* has minimum failing rate, which is 0.001. It is a fault under the category of “*Wrong logic or relational operators*” fault. According to [47], listed in Table 6.3, such a fault has a high chance to appear in realistic programs. Also, take this fault as an example. Table 6.3 is explained as follows. According to the *Orthogonal Detect Classification Class*, this fault belongs to the *Check* class, which occupies 26.6% portion among all classes. In this class, the fault is further under the category of *C2: Wrong logic or relational operators*. Such a subclass occupies 47% portion of this class.



6.5.2 Peer Techniques and Peer Methods

We use CBI and SOBER as two alternatives of TC1 techniques. We apply two parametric hypothesis testing methods, the Student's t-test and the F-test, and two non-parametric hypothesis testing methods, the Mann-Whitney test and the Wilcoxon signed-rank test, in our framework, and use them as alternatives of TC2 techniques and TC3 techniques, respectively. The details of these fault-localization approaches are explained as follows.

The Wilcoxon signed-rank test is a kind of non-parametric test used to compare two distributions [78]. In our experiment, we adopt the second distribution and the mean of the first distribution to conduct the Wilcoxon signed-rank test, and use the resultant p-value of hypothesis testing to calculate P-score in Equation (6.2). Thus, our method using the Wilcoxon signed-rank test (Wilcoxon for short) captures the difference of the evaluation bias for all passed executions and those for all failed executions. Similar to the Mann-Whitney test, the Wilcoxon signed-rank test is also selected as an alternative in our model.

The Student's t-test hypothesis testing method checks hypotheses about the fact that the means of two random variables, which are represented by two samples, are equal [78]. The F-test hypothesis testing method checks hypotheses about the fact that the dispersions of two random variables, which are represented by two samples, are equal [78]. Since both of them work correctly under the conditions that (i) both random variables have normal distributions and (ii) samples are independent, they belong to parametric hypothesis testing methods. Though there is no evidence that these two conditions can be always satisfied in our experiment, we directly apply these two parametric hypothesis testing methods in our experiment to investigate their effectiveness. We use their p-values as the ranking scores in the experiment.

The other three fault-localization approaches, CBI [74][75], SOBER [77][76], and the use of the Mann-Whitney test [62][119] have been introduced in previous sections.

6.5.3 Effectiveness Metrics

In this section, we first introduce our performance metrics, P-score, to measure effectiveness of fault-localization techniques. Then, we introduce a statistical normality test and discuss the use of its p-value as a metrics of normality. Finally, we introduce the correlation relation metrics.



Effectiveness Metrics: P-score

The metric t-score has been used in previous studies [36][76][77][90] to evaluate the effectiveness of predicate-based fault-localization techniques. It starts from some top prioritized predicate statements, adopts a Breath-First Searching manner, searches the whole space of statements, and then uses the percentage of statements examined, before reaching any faulty statement, as the result of effectiveness.

The t-score metrics helps measure the cost of locating a fault using a fault-localization technique. However, some limitations have been reported on the use of t-score in previous work [36][90]. (i) They claim that their evaluation setup “*assumes an ideal programmer who is able to distinguish defects from non-defects at each location, and can do so at the same cost for each location considered.*” [36]. (ii) Besides, t-score assumes that the developer can follow the control- and/or data- dependency relations among statements when searching for faults. However, there is no evidence that it resembles the manner of debugging in real life.

To better reflect the effectiveness of the non-parametric fault-localization technique, we propose a novel metrics, which we call **P-score**, to evaluate them. We recall that many fault-localization techniques [68][69][70][76][77] (including the non-parametric fault-localization technique) generate a predicate list, which contains all the predicates sorted in descending order of their degree of fault relevance (in terms of how much each of them is deemed to be relevant to fault). Such degree of fault relevance is measured by the ranking formula of the technique. For postmortem analysis, we mark the predicate closest (in terms of their positions, i.e., line number, in the code) to any fault in the program, and use the position of the predicate in the predicate list as the indicator of the effectiveness of a fault-localization technique in generating the predicate list. We call such a predicate *the most fault-relevant predicate*. Suppose L is the predicate list and \tilde{P} is the most fault-relevant predicate. The measurement formula is given by equation (6.2). To ease our presentation, we simply call this metrics the P-score.

$$P\text{-score} = \frac{1 - \text{based index of } \tilde{P} \text{ in } L}{\text{number of predicates in } L} \times 100\% \quad (6.2)$$

The metrics P-score reflects the effectiveness of a fault-localization technique. The lower the value, the more effective will be the fault-localization technique. For tie cases, which mean that there exist multiple most fault-relevant predicates on the same predicate list, we count \tilde{P} as the first one reached in L .

For example, the faulty version “v1” of program “schedule2” (from the Siemens programs) contains 43 predicates. The fault lies in line 135; the most fault-



relevant predicate exists in line 136. Suppose a fault-localization technique ranks predicate $\tilde{P} = P_{136}$ at the second top position in the generated predicate L . The P-score is calculated as $\frac{2}{43} \times 100\% \approx 4.65\%$.

The metric P-score uses the appearance position of the *most fault-relevant predicate* in the generated predicate list as the effectiveness of that fault-localization technique. It is expressed as Equation (6.2). The lower the value is, the more effective the fault-localization technique is. Note that if there exists more than one most fault-relevant predicate, in other words, if a tie case is encountered, we count \tilde{P} as the one having highest priority in L .

Normality Test: the Jarque-Bera Test

To measure whether the evaluation biases of a predicate form a normal distribution, we adopt the standard normality test method, the Jarque-Bera test [78]. The Jarque-Bera test is used to test the null hypothesis that the given population is from a normal distribution. The p-value of the Jarque-Bera test is used to measure how much the evaluation biases of a predicate form a normal distribution. For example, a p-value less than 0.05 means that the null hypothesis can be rejected at the 0.05 significance level [78]. It also means that the probability of obtaining an observation agreeing with the null hypothesis (being normal distribution) is less than 0.05. In general, the smaller the p-value, the more confident we will be in rejecting the null hypothesis. In other words, the smaller the p-value, the farther will be the evaluation biases of the predicate from a normal distribution.

To help readers follow the idea of normality test, we use three different populations to illustrate the outcomes of the Jarque-Bera test. We use histograms to represent the distributions of these three populations. The respective histograms are shown in Figure 6.4. We observe that, among the three populations, the leftmost one (Figure 6.4(a)) is closest to a normal distribution. The rightmost one (Figure 6.4(c)) is farthest from a normal distribution. The central one (Figure 6.4(b)) is in between the two scenarios. The result of the p-value for the population in Figure 6.4(a) is 0.7028. It means that we have a 70.28% probability that the observed data in Figure 6.4(a) is from a normally distributed population. The result of the p-value for the population in Figure 6.4(b) is 0.2439. It means that, we have a 24.39% probability that the observed data in Figure 6.4(b) is from a normally distributed population. The p-value result of the population in Figure 6.4(c) is 0.0940. According to the normality test results, we can determine that the population in Figure 6.4(a) is closest to a normal distribution, followed by the population in Figure 6.4(b), while the population in Figure 6.4(c) is farthest from



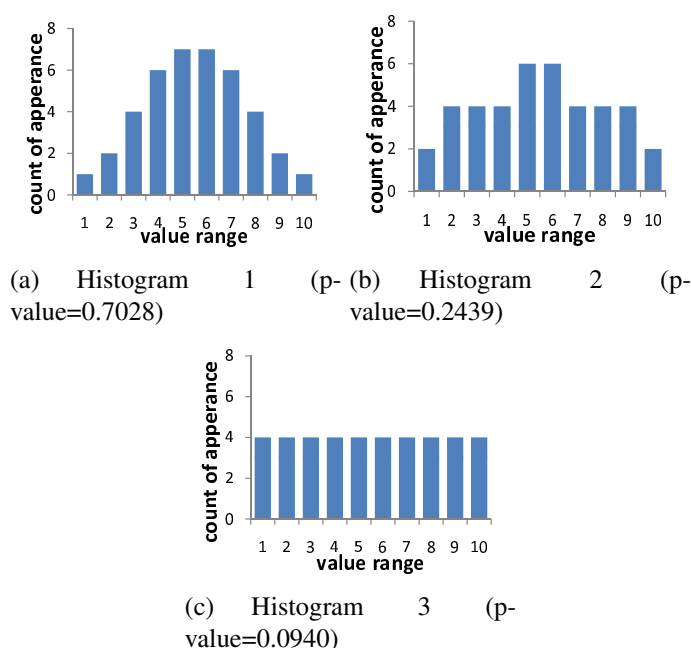


Figure 6.4: Illustration for normality test

a normal distribution. The results of the normality test match our expectation.

In the rest of this thesis, we will use the results of the p-value in the Jarque-Best test as the degree of normality for predicates.

Correlation Metrics: Pearson Correlation Test

Pearson Correlation test [78] is designed to evaluate the correlation coefficient of two populations. It is used to test the strength and direction of the linear relationship between two populations. The result of the Pearson Correlation test is in the range of $[-1, 1]$. The correlation is close to 1 in the case of an increasing linear relationship. It is close to -1 in the case of a decreasing linear relationship. If the two populations are independent of each other, the correlation is close to 0.

For example, we use three sets of data to illustrate the outcomes of the Pearson Correlation test. The three sets are represented by the points in Figures 6.5(a), 6.5(b), and 6.5(c), respectively. For each point, the X and Y coordinates stand for the values of the X and Y variables, respectively.

Let us first focus on the leftmost set of data (Figure 6.5(a)). The Y coordinate conforms to a linear increasing function of the X coordinate. The Pearson Cor-

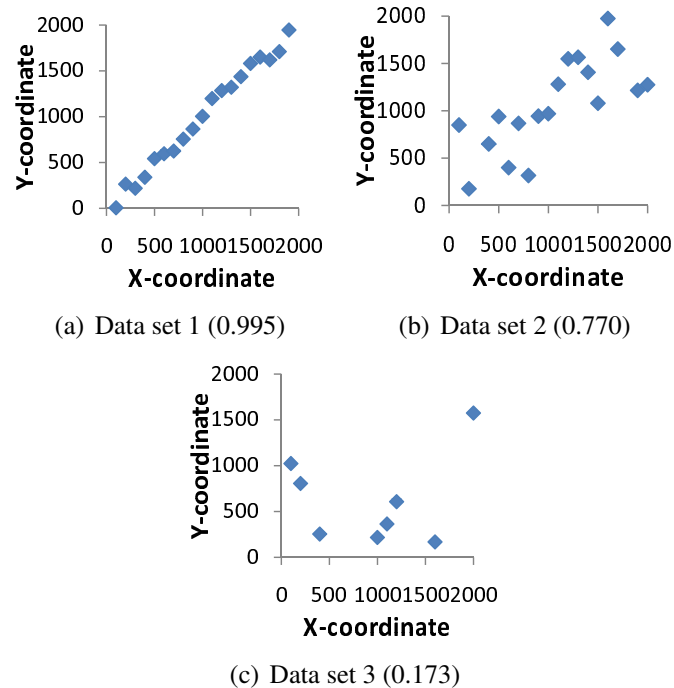


Figure 6.5: Illustration of Pearson Correlation test

relation result for this set of data is 0.995. For the rightmost set of data (Figure 6.5(c)), the X and Y coordinates do not have strong dependence relationships between each other. The Pearson Correlation test for this set of data is 0.173. For the set of data in Figure 6.5(b), the situation is in between the two scenarios above. The corresponding correlation test result is 0.770. From these examples, we observe that the Pearson Correlation test is useful in determining the correlation relationships between two populations.

6.5.4 Experimental Setup

Among the 132 programs, two of them (version “v27” of program “replace” and version “v9” of program “schedule2”) come with no failure-causing test cases. This is also reported in previous work [76][77]. These two versions are excluded because both our method and SOBER need the presence of both passed and failure-causing test cases. To evaluate our method, we follow [76][77] to use the whole test suite as input to our method and SOBER (except Section 6.5.5 and 6.5.5). Again, following [76][77], we use branches and returns (see Section 6.3.1) as program locations for predicates in the experiment.

For each of the 132 faulty versions, we manually identify the faulty statements by comparing the faulty version to the original version. For example, in the faulty version “v1” of program “tot_info” (Figure 6.1), statement E_1 is the faulty statement. If a fault lies in a global definition statement or it is a statement omission fault, we mark the directly affected statements or the next adjacent statement as the faulty statement(s).

For each of the 132 faulty versions, we manually mark the most fault-relevant predicate in them. For 111 out of 130 of them, there is no ambiguity to identify the most fault-relevant predicate. The most fault-relevant predicate is always 3 lines far from a faulty statement. For example, in the faulty version “v1” of program “tot_info” (Figure 6.1), predicate P_4 is the most fault-relevant predicate. For the rest 19 of them, faults exist in modules having no predicate, and the most fault-relevant predicate cannot be determined. We therefore exclude these faulty versions in our experiment. They are versions “v4”, “v5”, and “v6” of program “print_tokens”, version “v12” of program “replace”, versions “v13”, “v14”, “v16”, “v17”, “v18”, “v19”, “v33”, “v36”, “v38”, “v7”, and “v8” of program “tcas”, and versions “v10”, “v12”, “v21”, and “v6” of program “tot_info”.

We conduct our experiment using a Dell PowerEdge 1950 server running a Solaris UNIX with kernel version Generic 120012-14. The tools used to build up our experimental platform include flex++ 2.5.31, bison++ 1.21.9-1, CC 5.8. The implementation of the two standard non-parametric hypothesis testing methods “the Mann-Whitney test” and “the Wilcoxon signed ranked test”, and the two standard parametric hypothesis testing methods “the Student’s t-test” and “the F-test” are downloaded from the ALGLIB website (available at <http://www.alglib.net/>).

6.5.5 Results and Analysis

In this section, we analyze the results from several dimensions.

Overall Effectiveness Comparison

Figure 6.6 shows the results of using P-score to evaluate the effectiveness of the six techniques (Wilcoxon, Mann-Whitney, CBI, SOBER, t-test, and F-test). It depicts the percentage of faults that can be located when a certain percentage of predicates are examined. The curves labeled as “Wilcoxon” and “Mann-Whitney” show the results of Wilcoxon and the result of Mann-Whitney, respectively. The curves labeled as “CBI” and “SOBER” show the results of CBI and SOBER, respectively. The curves labeled as “t-test” and “F-test” show the re-



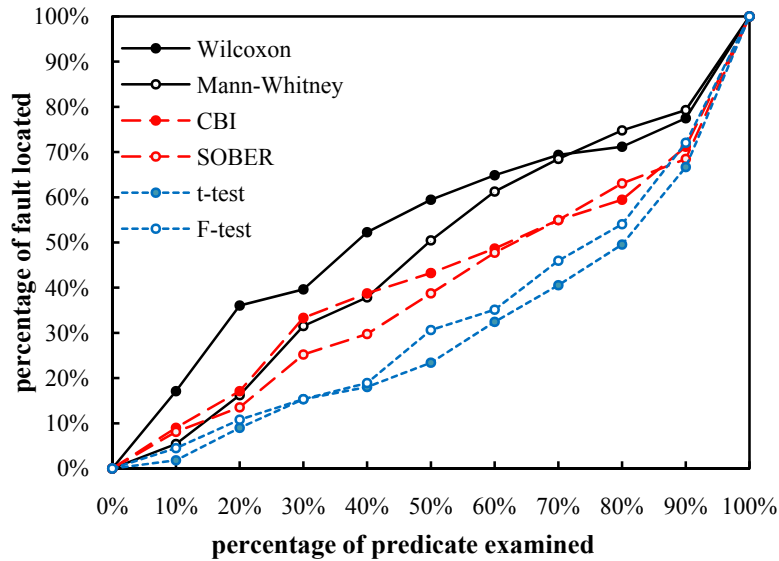


Figure 6.6: Overall effectiveness comparisons

sults of t-test and the result of F-test, respectively. The target faulty versions are the 111 faulty versions with the faulty statement(s) identified and the most fault-relevant predicate marked out.

Let us first take the predicate examining points 10% and 20% to illustrate. When examining up to 10% of all the predicates in the generated predicate list, Wilcoxon and Mann-Whitney can reach the most fault-relevant predicate in 17.12% and 5.41% of all the 111 faulty versions, respectively; CBI and SOBER can reach the most fault-relevant predicate in 9.01% and 8.11% of all the 111 faulty versions, respectively; while t-test and F-test can reach the most fault-relevant predicate in 1.80% and 4.50% of all the 111 faulty versions, respectively. When examining up to 20% of all the predicates in the generated predicate list, Wilcoxon and Mann-Whitney can reach the most fault-relevant predicate in 36.04% and 16.22% of all the faulty versions, respectively; CBI and SOBER can reach the most fault-relevant predicate in 17.12% and 13.51% of the 111 faulty versions, respectively; while t-test and F-test can reach the most fault-relevant predicate in 9.01% and 10.81% of the 111 faulty versions, respectively.

In the range of [10%, 80%], both CBI and SOBER outperform t-test and F-test. In the range of [10%, 90%], Wilcoxon always outperforms CBI and SOBER, while the effectiveness of Mann-Whitney is comparable to (in the range of [10%, 40%]) or better than (in the range of [50%, 90%]) CBI and SOBER. From this figure, we observe that Wilcoxon performs better than CBI and SOBER.

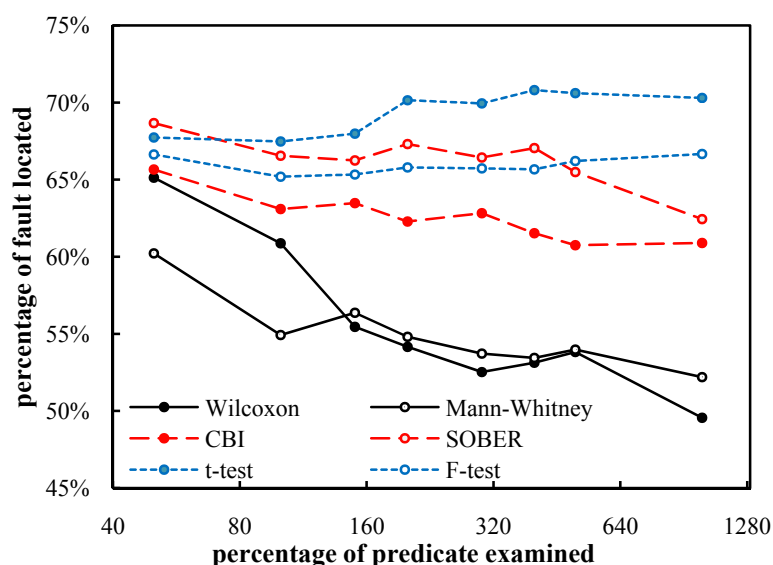


Figure 6.7: Effect of test suite size (lower the curve, better the technique)

Mann-Whitney performs better than, or if not, comparable to, CBI and SOBER; while CBI and SOBER perform better than t-test and F-test.

Scalability Test in Terms of Test Suite Size

In this section, we study the effect of test suite size on fault-localization techniques. Briefly speaking, we change the size of the test suite, monitor the effectiveness of fault-localization techniques, and plot the trend of changing of effectiveness.

Figure 6.7 shows the effect of test suite size on fault-localization techniques. The X-axis stands for the test suite size, which is controlled from 50 to 1000. Limited by the effort we afforded, the suite sizes chosen are 50, 100, 150, 200, 300, 400, 500, and 1000. (In the controlled experiment, the test cases are randomly selected from the test pool.) The Y-axis stands for the mean percentage of predicates examined to locate the most fault-relevant predicate. The target programs are the 111 faulty versions with the most fault-relevant predicate marked. The effectiveness metric used is P-score. The curves with labels are similarly explained as in previous figures.

From Figure 6.7, we observe that, as the size of test suite increases, all the six curves show decreasing trends. It means that the effectiveness of these fault-localization techniques increases when inputting a test suite with larger size. We

believe that it may be due to the statistical nature of these techniques. It is well known that the robustness of statistical methods increase as the size of sample set increases. Another observation is that the trend of decreasing of effectiveness for CBI, SOBER and Mann-Whitney are slower than those of Wilcoxon, as the size of sample set increases. At the same time, the trend of decreasing of effectiveness for t-test and F-test are slower than those of CBI, SOBER, and Mann-Whitney, as the size of sample set increases. The results also show that using Wilcoxon or Mann-Whitney in our model is more effective for test suite having larger number of test cases than for small sized test suite.

Though CBI and SOBER use self-proposed hypothesis testing models, their scalability is better than using any of the two parametric hypothesis testing models in our framework. This is because the models used in CBI and SOBER are driven by their fault-localization heuristics and have advantages than a general parametric hypothesis testing model. However, the scalability of using both of the two non-parametric hypothesis testing models in our framework is better than CBI and SOBER. This because their presumption of normal distributed feature spectra do not fit the realistic environment, and thus introduces inaccuracy to their model.

Efficiency Analysis

In this section, we report the efficiency of our implementation of these fault-localization techniques.

Figure 6.8 shows the program names, and the mean execution time of using these techniques to rank the predicates. The target programs are the 111 faulty versions. The test suite size is 1000. All the times spent are collected by sequentially executing each technique to rank the predicates in each faulty version. The six categories represent the overall results, and the results on “replace”, “print_tokens” and “print_tokens2”, “schedule” and “schedule2”, “tcas”, and “tot_info”, respectively. In each category, the six different bars respectively shows the mean running time of the six techniques on the faulty versions of that program category.

Let us take the first category in Figure 6.8 as illustration. In it there are six bars, which in order stand for the mean time spent of Wilcoxon (0.544 seconds), Mann-Whitney (0.387 seconds), CBI (0.613 seconds), SOBER (0.605 seconds), t-test (0.400 seconds), and F-test (0.397 seconds), to rank the predicates in the 111 faulty versions.

From Figure 6.8, we observe that the running time show an increasing trend as the program scales increase. For example, programs “replace”, “print_tokens”,



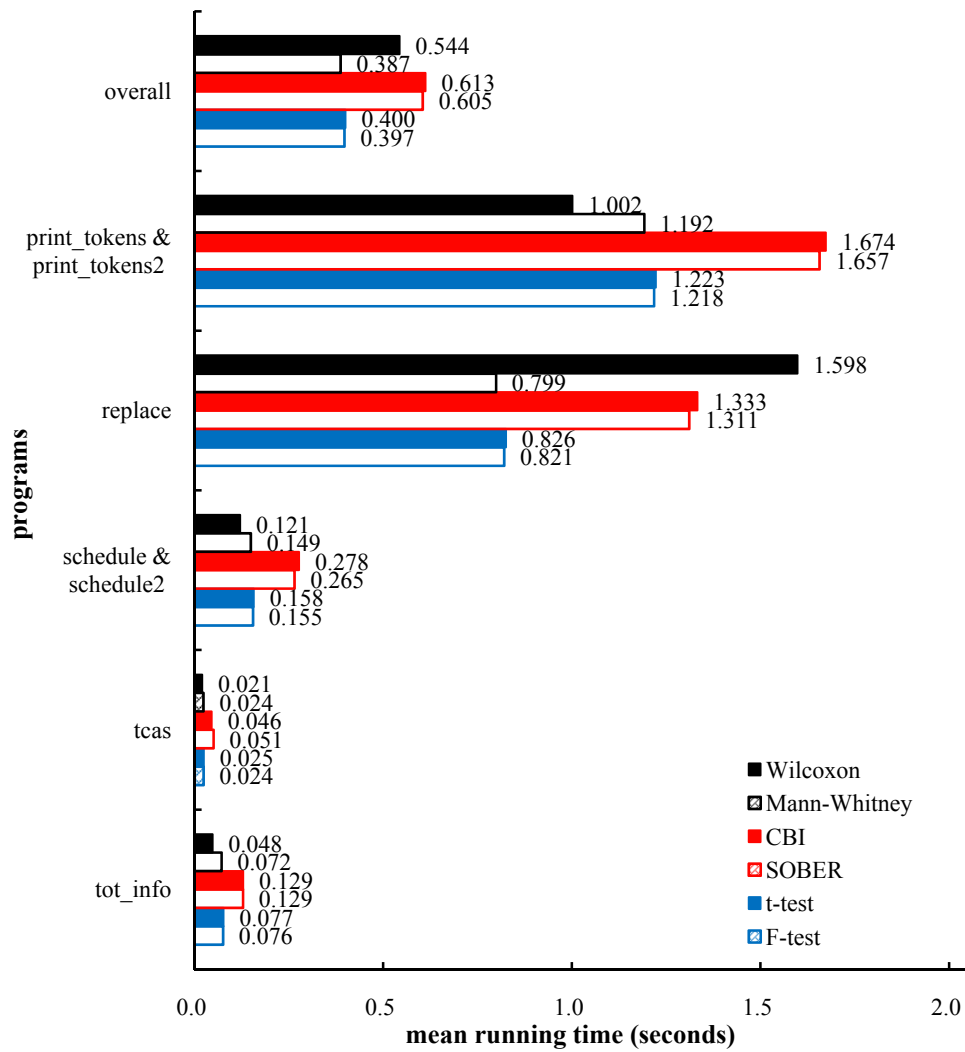


Figure 6.8: Time spend of each techniques on different programs

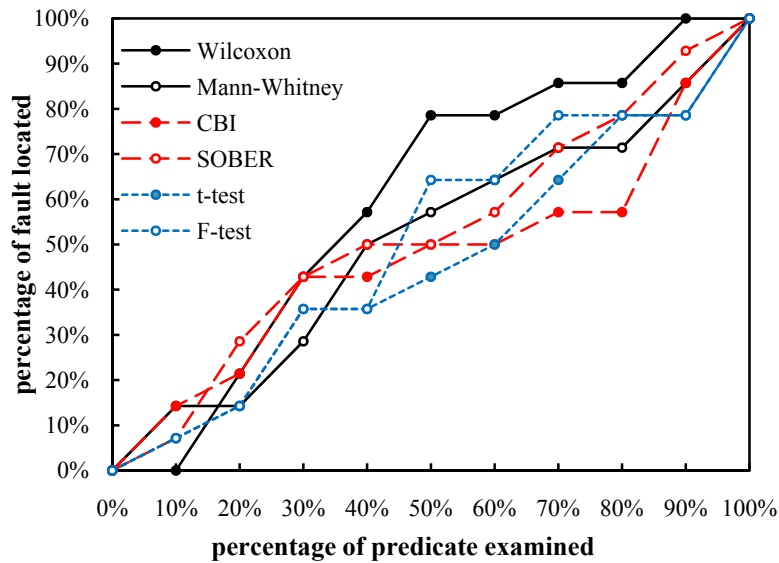


Figure 6.9: Effectiveness comparisons on `print_tokens` & `print_tokens2`

and “`print_tokens2`” have larger scales than programs “`schedule`”, “`schedule2`”, “`tcas`”, and “`tot_info`”, and the mean running time of each technique in the former three programs is longer than those in the latter four programs. This is understandable. It can be explained as the larger the program scale is, the larger the program contains more predicates. The more predicates need to be processed, the more time is consumed.

We also observe that, compared to the running time of CBI and SOBER, the running time of Wilcoxon, Mann-Whitney, t-test, and F-test are also acceptable. It means that the techniques generated using our model are feasible in practice.

Individual Effectiveness Comparison

In this section, we report the results of these fault-localization techniques on each individual program.

Figure 6.9 to 6.13 show the results of these techniques on individual Siemens programs. The program `print_tokens` has only 4 faulty versions and cannot form a meaningful sample set. We combine them with the faulty versions of `print_tokens2`, and show them together in Figure 6.9. Such a consideration is also based on the fact that these two programs have similar program structures. For the same reason, we combine the faulty versions of program `schedule` and `schedule2`, and show them in Figure 6.11. Figure 6.10, 6.12, and 6.13 shows the

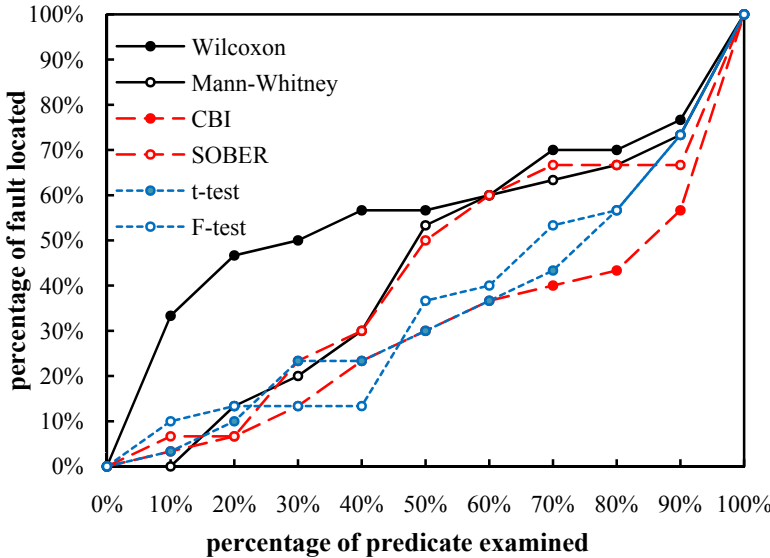


Figure 6.10: Effectiveness comparisons on replace

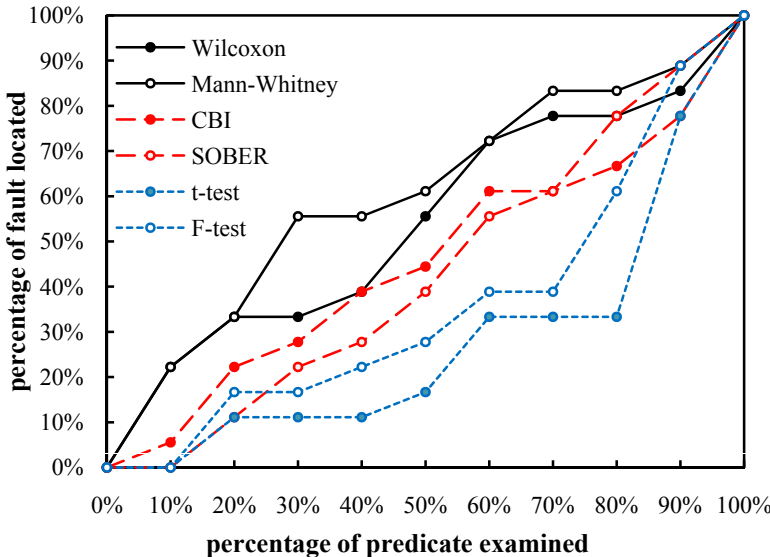


Figure 6.11: Effectiveness comparisons on schedule & schedule2



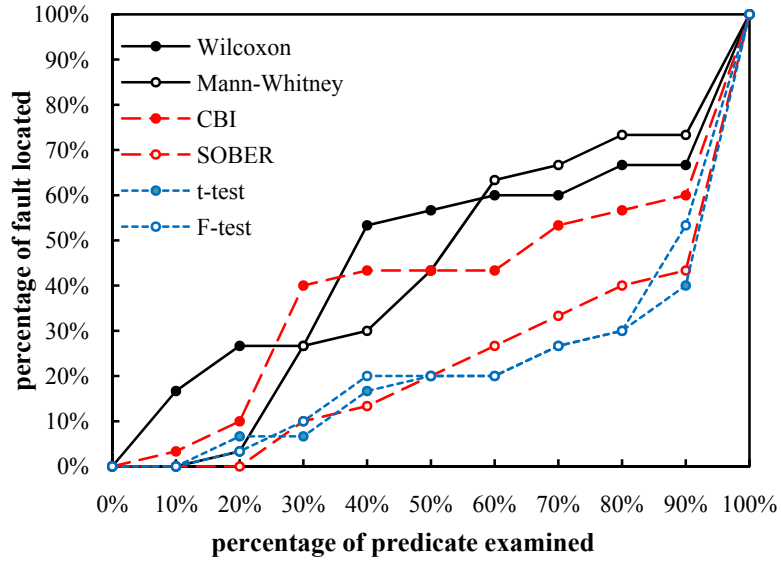


Figure 6.12: Effectiveness comparisons on tcas

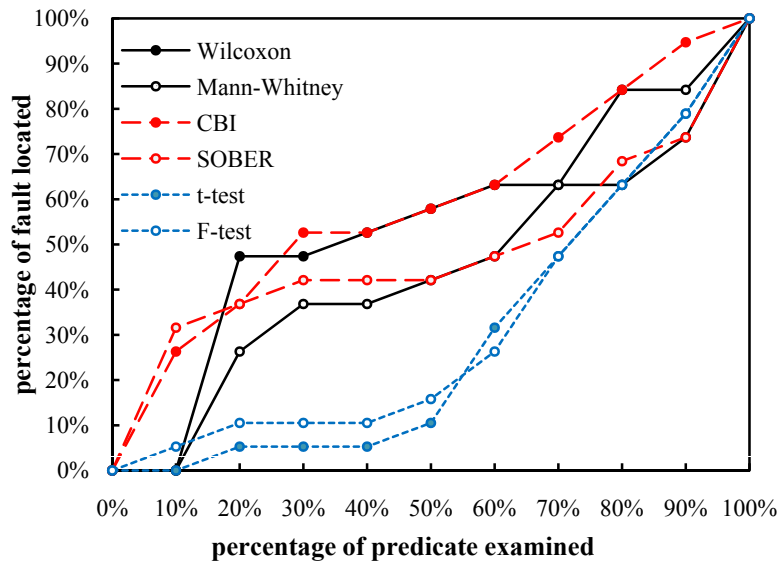


Figure 6.13: Effectiveness comparisons on tot_info



results on faulty versions of program `replace`, `tcas`, and `tot_info`, respectively.

Let us first focus on Figure 6.9, when checking up to 30% of all the predicates, none of the six techniques show obvious advantages to the others. However, in the rest of the predicate examining range, Wilcoxon is always the most effective one. The other five techniques perform comparably.

In Figure 6.10, the curve of Wilcoxon is always above each of the other five curves. It means that the technique of using the Wilcoxon signed-rank test in our model is the most effective one among these six techniques. The curve of Mann-Whitney and the curve of SOBER are close to each other. It means that Wilcoxon and SOBER are comparably effective. The other these techniques, CBI, t-test, and F-test, are comparably effective with one another.

In Figure 6.11, Mann-Whitney and Wilcoxon perform more effectively than CBI and SOBER, except that, in the range of [80%, 100%], SOBER catches up with Wilcoxon. In the majority of the range, say [30%, 80%], CBI and SOBER are more effective than t-test and F-test. In Figure 6.12, Wilcoxon, Mann-Whitney, and CBI are comparably more effective than the other three techniques. In Figure 6.13, Wilcoxon, Mann-Whitney, CBI, and SOBER performs comparably. Their effectiveness has great advantages over that of t-test and F-test.

In summary, our intuitive observations are: (i) Wilcoxon and Mann-Whitney are more effective, or if not, as effective as CBI and SOBER, (ii) CBI and SOBER are more effective than t-test and F-test in most cases.

Answering Research Questions Q1 - Q5

In previous section, we have the observation that TC3 techniques are more effective than TC1 techniques and TC1 techniques are more effective than TC2 techniques, in most cases. To figure out whether there exist significant differences among these techniques and enhance our intuitive observation, we conduct hypothesis testing as follows.

To answer the research questions in Section 6.4, we collect the effectiveness of these six techniques on the 111 faulty versions, and conduct hypothesis testing to determine the relative order of effectiveness for each pair of techniques. The results are shown in Table 6.4. Take the right-most number “0.0068” as illustration, it means the p-value of hypothesis “F-test has significant improvements on t-test” is 0.0068. Here, the null hypothesis is given as “F-test does not have significant improvements on t-test”, which can be rejected as 5% significant level. Note that in the table, we do not show those numbers that are greater than 0.05. Those cells, which corresponding numbers are greater than 0.05, are



Table 6.4: The p-value results of hypothesis “technique X has significant improvements on technique Y ”

$Y =$	$X =$				
	Wilcoxon	Mann-Whitney	CBI	SOBER	F-test
t-test	< 0.0001	< 0.0001	= 0.0002	< 0.0001	= 0.0068
F-test	< 0.0001	= 0.0003	= 0.0074	= 0.0080	
SOBER	< 0.0001	= 0.0362			
CBI	= 0.0025				
Mann-Whitney	= 0.0327				

leave blank. Therefore, the cells with number mean that the corresponding technique X outperforms the corresponding technique Y (the null hypothesis can be rejected at 5% significant level). The cells without number filled in mean that the corresponding technique X has no significant difference with that of the corresponding technique Y .

Finally, we summarize the hypothesis testing results as,

- R1: Technique Wilcoxon significantly outperforms Mann-Whitney, Mann-Whitney significantly outperforms SOBER, and SOBER in turn significantly outperforms F-test,
- R2: Technique Wilcoxon significantly outperforms CBI, and CBI significantly outperforms F-test,
- R3: Technique F-test significantly outperforms t-test.

Therefore, we can answer the research questions in Section 6.4 as,

- A1: Comparing to TC1 techniques, TC2 techniques are not more effective.
- A2: Comparing to TC2 techniques, TC3 techniques are more effective?
- A3: Comparing to TC1 techniques, not both TC2 and TC3 techniques are more effective.

Further, we also conduct hypothesis testing to answer the other research questions in Section 6.4. For each curve in Figure 6.7, we compute the difference between two adjacent check points, and compare it with a series of zero. Thus, we conduct hypothesis testing (the Student’s t-test is used) to test the hypothesis



that “the effectiveness of technique in test has a significant improvement with the increasing of test suite size”. The corresponding null hypothesis is given as “the effectiveness of technique in test has no significant improvement with the increasing of test suite size”. The hypothesis testing results for each of the Wilcoxon, Mann-Whitney, t-test, F-test, SOBER, and CBI are 0.05, 0.21, 0.33, 0.99, 0.16, and 0.16, respectively. If we deem 0.05 as the significance level, the effectiveness of the technique Wilcoxon is deemed to have a significant improvement with the increasing of test suite size. If we use their resultant p-value from their hypothesis testing as a measure to compare their scalability, in terms of test suite size. Our conclusion is that Wilcoxon has the best scalability, and in turn, CBI and SOBER, Mann-Whitney, t-test, and F-test. As a result, we give the answer to aforementioned research question *Q4*.

A4: With the increasing of test suite size, T2 techniques do not gain more effective fault-localization results; while some T3 techniques gain more effective fault-localization results.

In our efficiency test report (Figure 6.8), there are too limited samples to form statistically meaningful hypothesis testing. Therefore, we give intuitive observation and conclude that they have comparably practical running time (the running time is always at a second level). As previous, we answer the final research question.

A5: In terms of running time, T1, T2, and T3 techniques have comparably practical efficiencies.

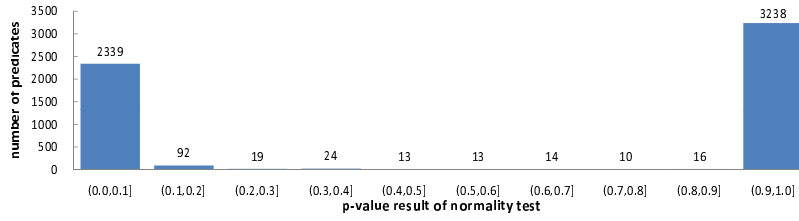
In summary, we conclude that, over the Siemens programs, use of non-parametric hypothesis testing methods improves the effectiveness of previous predicate-based fault-localization techniques.

Answering Research Question *Q6*

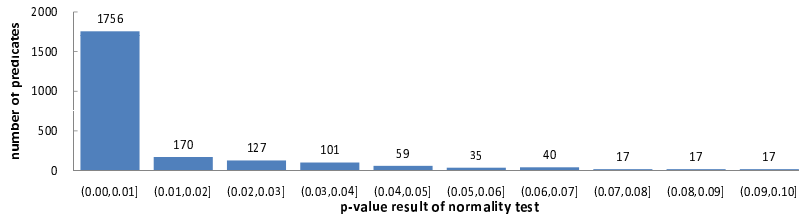
In this section, we first conduct normality test on the evaluation biases of 5778 predicates from 111 faulty versions of Siemens programs, and report the distribution of evaluation biases for these predicates. After that, we use hypothesis testing method to answer *Q6*.

The distributions (represented in form of histograms) of the normality test results are shown in Figure 6.14. For each predicate, we separately consider its evaluation biases in passed executions and its evaluation biases in failed executions. For each predicate, we specify its normality test result as the minimum of its normality test result of evaluation biases in all passed executions and that in

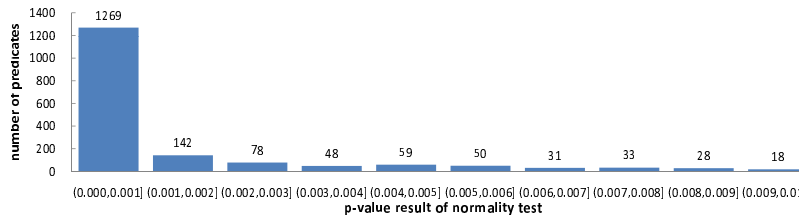




(a) range of 0 to 1



(b) range of 0.0 to 0.1



(c) range of 0.00 to 0.01

Figure 6.14: Results of normality test for predicates

all failed executions. Figure 6.14(a) shows the number of predicates that have corresponding p-value for normality test of their evaluation biases, in the range of $[0, 1]$ (10 segments). The leftmost data column stands for the predicates having p-values (of normality test for their evaluation biases) less than or equal to 0.1. The number of such predicates is 2399. It means that, if the null hypothesis (evaluation biases of predicates form normal distributions) is true, for 2399 predicates, the probability of the appearance of their observed evaluation biases is less than 10%. In other words, for these 2399 predicates, at the 10% significance level, the null hypothesis can be rejected. The rightmost column stands for the predicates having p-values greater than 0.9. The number of such predicates is 3238. It means that, if the null hypothesis is true, for these 3238 predicates, the probability of observing the sample evaluation biases is higher than 90%. The central eight columns respectively mean the predicates having p-values in ranges



of $(0.1, 0.2]$, $(0.2, 0.3]$, $(0.3, 0.4]$, $(0.4, 0.5]$, $(0.5, 0.6]$, $(0.6, 0.7]$, $(0.7, 0.8]$, and $(0.8, 0.9]$. These ranges are decided by uniformly dividing the range of $[0, 1]$. The second plot and third plot show the same statistics in ranges of $[0.0, 0.1]$ and $[0.00, 0.01]$, respectively.

From Figure 6.14(a), we observe that not all of the predicates form statistically meaningful normal distributions. If we choose 0.1 as the significance level for the p-value of normality test, the null hypothesis can be rejected for 2339 (more than 0.4) predicates. We deem their evaluation biases to be far from having normal distributions. If we deem 0.9 as the significance level for the p-value of normality test, 3238 (less than 60%) of them are recognized to have normally distributed evaluation biases. We deem that their evaluation biases have normal distributions.

Figure 6.14(b) is a zoom-in representation of the range of $[0.0, 0.1]$ (the range of the leftmost data column of Figure 6.14(a)). We notice that, for 1756 out of 2339 (more than 75%) of the predicates having p-values in the range of $[0.0, 0.1]$, their corresponding p-values concentrate in the range of $[0.00, 0.01]$. Let us focus on Figure 6.14(c). We further notice that, for 1269 out of 1756 (near 75%) of the predicates having p-values in the range of $[0.00, 0.01]$, their p-values concentrate in the range of $[0.000, 0.001]$.

Our observation is that: (i) For no more than 60% of the predicates, their evaluation biases have normal distributions (the null hypothesis cannot be rejected at a significance level of less than 0.9). (ii) For a majority of the rest 40% of the predicates, their evaluation biases are far from having normal distributions (the null hypothesis can be rejected at the 0.1 significance level). (iii) There are few predicates whose degrees of normality are within the range of $[0.1, 0.9]$.

Recall that we make use of the standard normality test method, the Jarque-Bera test, to measure the *degree of normality* for the evaluation biases of predicates. The higher the degree of normality (p-value result of the Jarque-Bera test) for a predicate is, the closer the observed evaluation biases is to a normal distribution. Since we calculate a degree of normality for each predicate, we design the following null hypothesis to answer the research question of $Q6$:

H1: “The mean degree of normality for the tested predicates is greater than a given threshold θ_1 .”

Such a null hypothesis captures the *mean* degree of normality for the predicates, and is therefore used to determine whether normal distributions are *common* for predicates. Besides, we introduce a parameter θ_1 to control the power of the null hypothesis. The higher the value of θ_1 is, the more confidence the null



hypothesis has. Therefore, for same set of predicates, the higher a θ_1 is chosen, the easier the null hypothesis can be rejected.

To answer research question Q_6 , we conduct the one-tail Student's t-test [125] to test H_1 . The p-value of the Student's t-test means that by how much probability is the observed predicates from a population with mean degree of normality greater than θ_1 . For example, suppose we have only three predicates. The degrees of normality for them are 0.640, 0.750, and 0.860, respectively. The p-value result of the Student's t-test on null hypothesis H_1 with $\theta_1 = 0.600$ will be 0.929. On the other hand, the p-value result of the Student's t-test on null hypothesis H_1 with $\theta_1 = 0.750$ will be 0.500. Similarly, the p-value result of the Student's t-test on null hypothesis H_1 with $\theta_1 = 0.900$ will be 0.071. It means, equally, the probability that the predicates are from a population with mean degree of normality greater than 0.600 is 92.9%. The other two numbers are similarly explained.

We vary the value of θ_1 in the range of $[0, 1]$. The corresponding p-value of one-tail Student's t-test results are shown in Table 6.5. The upper row shows the threshold values for θ_1 , the lower row shows the one-tail Student's t-test result in terms of p-value.

threshold value (θ_1)	0.000 - 0.500	0.576	0.584	0.587	0.591	0.600 - 1.000
p-value result	1.000	0.500	0.100	0.050	0.010	≤ 0.0001

Table 6.5: Student's t-test on different threshold for H_1

We observe that we have great confidence (probability near to 100.0%) that these predicates are from a population with mean degree of normality greater than 0.500. At the same time, the probability that these predicates are from a population with mean degree of normality greater than 0.600 is less than 0.01%. Therefore, from the meaning of the null hypothesis and the symmetry of the one-tail test, we conclude that it is very likely (near to 100.0%) that these predicates are from a population with mean degree of normality in the range of $[0.500, 0.600]$.

We note that in statistics, in order to be statistically significant (to reject a null hypothesis), generally it requires at least 0.1 significance level [125]. Since we want to study the normality of program spectra in a conservative manner, we

set the threshold of degree of normality to a reasonable value (e.g., 0.700, 0.800, or higher) in the above null hypothesis. With $\theta_1 > 0.600$, the null hypothesis $H1$ can be always rejected at the 0.0001 significance level (the resultant p-value is less than 0.0001). Obviously, 0.0001 is a reasonably small value for significance level; and we conclude that normal distributions are *not* common for the evaluation biases of predicates. The answer to $Q6$ is *no*.

By the nature of null hypothesis, we know that the smaller the value of the threshold is, the more difficult we can reject the null hypothesis. For example, if we choose a value of 0.600 as the threshold. The hypothesis becomes: “*The mean degree of normality for the tested predicates is greater than 0.600*”. Still, such a null hypothesis can continue to be rejected at the 0.0001 significance level. Our results as presented in Table 6.5 indicates that many predicates cannot produce dynamic behaviors that form normal distributions even if one wishes to lower the judgment standard to wishfully assume that it could be the case.

Answering Research Question $Q7$

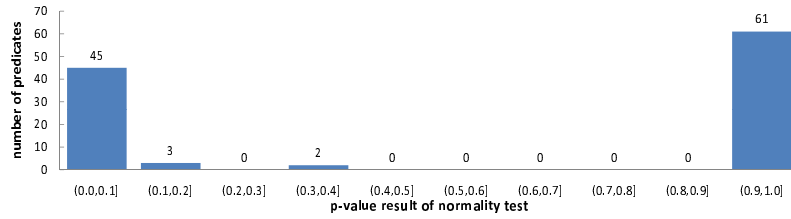
In this section, we first conduct normality test on the evaluation biases of the most fault-relevant predicates from the 111 faulty versions of Siemens suite, and report their normality distribution. After that, we use hypothesis testing method to study $Q7$. At the end of this subsection, we compare the statistical nature of the most fault-relevant predicates with the other predicates.

We know from the previous subsection that not all the predicates can be regarded as having normally distributed evaluation biases. One may wonder whether parametric technique may reasonably be applied if the distributions of the dynamic behavior of the most fault-relevant predicates are closely related to normal distribution. Therefore, in this subsection, we are going to study whether the evaluation biases of their most fault-relevant predicates may form normal distributions. Like the analysis in the previous subsection, the same normality test on the most fault-relevant predicates (111 in total) is applied. For each of these predicates, we separately consider its evaluation biases in passed executions and its evaluation biases in failed executions. Furthermore, for each predicate, we specify its normality test result as the minimum among its normality test result of evaluation biases in all passed executions and that in all failed executions. The results are shown in Figure 6.15.

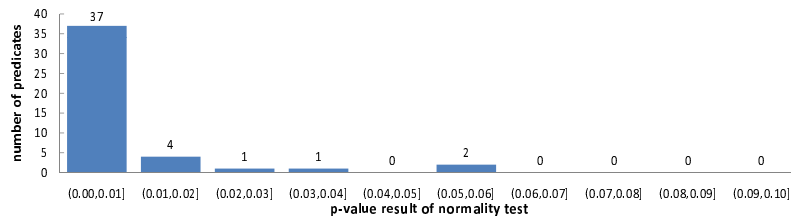
Figure 6.15 shows the distribution of the p-value results for normality test on the 111 most fault-relevant predicates in these faulty versions. It can be interpreted similar to Figure 6.14.

Let us focus on Figure 6.15(a) first. If we deem 0.9 as the significance level

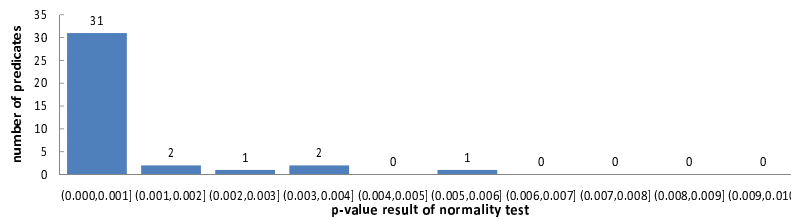




(a) range of 0 to 1



(b) range of 0.0 to 0.1



(c) range of 0.00 to 0.01

Figure 6.15: Results of normality test for the most fault-relevant predicates

for the p-value of normality test, 61 most fault-relevant predicates out of 111 (less than 60%) are recognized to exhibit normally distributed evaluation biases. It means that, if the null hypothesis is true, for 61 predicates, there is more than 90% probability for their observed evaluation biases to appear. On the other hand, if we choose 0.1 as the significance level (which is the de facto approach to apply hypothesis testing approach), there are still 45 predicates (more than 40%) that the null hypothesis can be rejected for them.

From Figure 6.15(b), we observe that, for 37 predicates out of 45 (more than 80%) having corresponding p-values in the range of $[0.0, 0.1]$, their p-values concentrate in the range of $[0.00, 0.01]$. When we zoom in further, as shown in Figure 6.15(c), 31 predicates out of 37 (near to 85%) which p-values are in the range of $[0.00, 0.01]$, their p-values concentrate in the range of $[0.000, 0.001]$.

Our observations are as follows: (i) For about 60% of the predicates, their



evaluation biases have normal distributions (the null hypothesis cannot be rejected at a significance level of less than 0.9). (ii) For a majority of the rest 40% of the predicates, their evaluation biases are far from having normal distributions (the null hypothesis can be rejected at the 0.1 significance level). (iii) There are few predicates whose normality test results are within the range of $[0.1, 0.9]$.

p-value range for normality test	>0.900	>0.500	>0.100	>0.050	>0.010
% of all predicates (number)	56.04% (3238)	58.69% (3391)	59.52% (3439)	61.70% (3565)	78.04% (4509)
% of the most fault-relevant predicates (number)	54.95% (61)	54.95% (61)	59.46% (66)	61.26% (68)	66.67% (74)

Table 6.6: Comparison of statistics of predicates with statistics of the most fault-relevant predicates

We further compare the statistics of the most fault-relevant predicates with the statistics of all predicates collected in the experiment. The results are shown in Table 6.6. Take the leftmost data column for example. It means that 56.04% (3238 out of 5778) of all studied predicates have p-value results of normality test greater than 0.900. However, only 54.95% (61 out of 111) of the most fault-relevant predicates have p-values greater than 0.900. We observe that the former value is higher than the latter value. The other columns show the similar phenomena. From this table, we observe that the degree of normality for the most fault-relevant predicates is generally lower than that for all predicates. This finding is important because it may indicate that effectiveness of existing parametric statistical fault-localization technique may be related to some unknown factors rather than the normality assumption that underlying such techniques; on the other hand, the study of such unknown factors is not within the scope of this thesis.

Similar to $Q6$, we design the following null hypothesis to answer the research question of $Q7$:



H2: “The mean degree of normality for the tested most fault-relevant predicates is greater than a given threshold θ_2 .”

Such a null hypothesis captures the *mean* degree of normality for the most fault-relevant predicates, and is therefore used to determine whether normal distributions are *common* for the most fault-relevant predicates. A parameter θ_2 is introduced to control the power of the null hypothesis. The higher the value of θ_2 is, the more confidence the null hypothesis has. Therefore, for same set of predicates, the higher a θ_2 is chosen, the easier the null hypothesis can be rejected.

To answer research question *Q7*, we also conduct the one-tail Student’s t-test [125] to testify *H2*. The p-value of the Student’s t-test means that by how much probability is the observed predicates from a population with mean degree of normality greater than θ_2 . We change the value of θ_2 in the range of $[0, 1]$. The corresponding p-value of one-tail Student’s t-test results are shown in Table 6.7. The upper row shows the threshold values, the lower row shows the one-tail Student’s t-test result in terms of p-value.

threshold value (θ_2)	0.000 - 0.400	0.561	0.621	0.638	0.671	0.700 - 1.000
p-value result	1.000	0.500	0.100	0.050	0.010	≤ 0.0001

Table 6.7: Student’s t-test on different threshold for *H2*

We observe that we have great confidence (probability near to 100.0%) that these predicates are from a population with mean degree of normality greater than 0.400. At the same time, the probability that these predicates are from a population with mean degree of normality greater than 0.700 is less than 0.01%. Therefore, from the meaning of the null hypothesis and the symmetry of the one-tail test, we draw the conclusion that it is very possible (near to 100.0%) that these predicates are from a population with mean degree of normality in the range of $[0.400, 0.700]$.

Similarly, since we want to study the normality of program spectra in a conservative manner, we set the threshold of degree of normality to a reasonable value (e.g., 0.700, 0.800, or higher) in the above null hypothesis. With $\theta_2 > 0.700$, the null hypothesis *H2* can be always rejected at the 0.0001 signif-

icance level (the resultant p-value is less than 0.0001). Since 0.0001 is a reasonably small value for significance level, we conclude that normal distributions are *not* common for the evaluation biases of the most fault-relevant predicates. The answer to *Q7* is *no*. And our results as presented in Table 6.7 indicates that many fault-relevant predicates cannot produce dynamic behavior that form normal distributions even if one wishes to lower the judgment standard to wishfully assume that it could be the case.

Answering Research Question *Q8*

In this section, we first report the finding and then analyze the result using hypothesis testing to answer *Q8*.

From previous subsections, we know that the assumption of evaluation biases of predicates forming normal distribution is not well supported by the experiment on the Siemens suite. Since the non-parametric fault-localization technique is supposedly not based on such an assumption, we predict that the effectiveness of non-parametric fault-localization technique does not correlate to the normality of predicates. Figure 6.16 gives the results of corresponding correlation tests. To investigate whether it is the case, we analyze the P-score of the predicate list produced by our fault-localization technique against the degree of normality.

Figure 6.16 depicts the correlations between our non-parametric fault-localization technique and the p-value of the most fault-relevant predicates. In this figure, there are 111 points, which stand for the 111 faulty versions. The X-coordinates mean the p-value results of normality tests for the most fault-relevant predicates. The Y-coordinates mean the P-score results for the same faulty version. This figure is divided into two parts. The left rectangle represents 55 most fault-relevant predicates with p-value less than 1.0. The right axis represents the 56 most fault-relevant predicates with p-value equal to 1.0.

We observe that, for the 56 points which evaluation biases form normal distributions (with p-value of value 1.0), their corresponding points distribute along the axis; for the rest 55 points which corresponding p-value is less than 1.0, they scatter within the rectangle on the left. We observe that as the p-value changes from 0.0 to 1.0, the P-score result of the fault-localization technique on the corresponding faulty version does not show an obvious increasing or decreasing trend. On the contrary, those P-score results appear scattering across the whole rectangle. Apparently, as far as our non-parametric fault-localization technique, the normality of evaluation biases for the most fault-relevant predicate does not strongly correlate to the effectiveness of the technique to locate faults.

We design the following hypothesis to answer *Q8*:



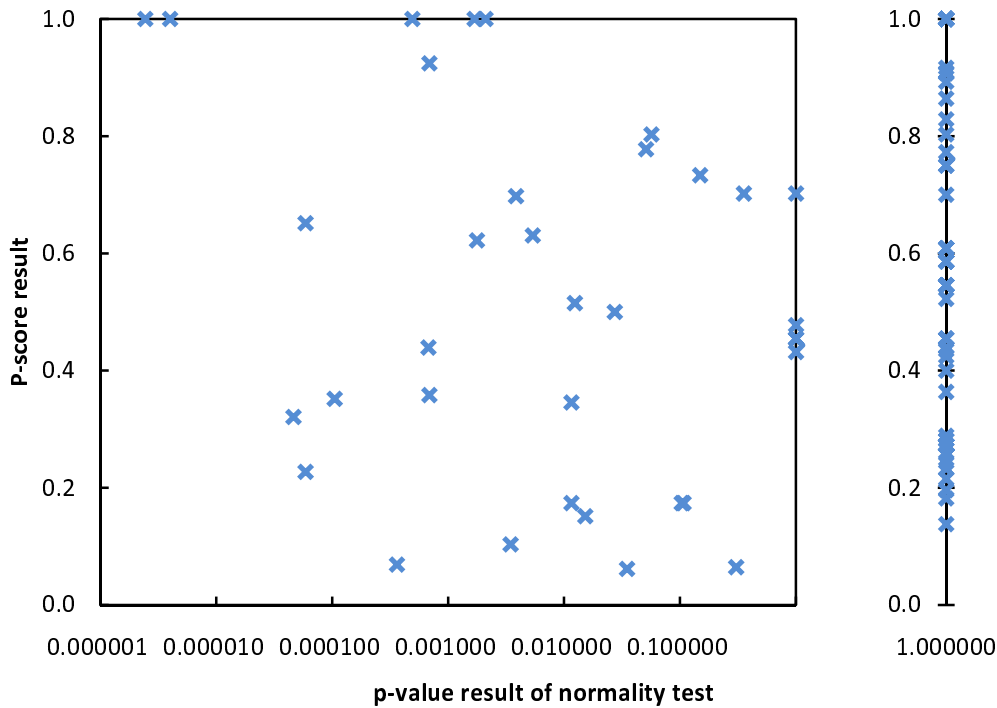


Figure 6.16: Effect of normality on fault-localization techniques

H3: “The correlation of normality and effectiveness is greater than a given threshold θ_3 .”

To scientifically investigate the effect of normality on these fault-localization techniques, we conduct the standard Pearson Correlation test on them. The Pearson Correlation test of normality and P-score for the non-parametric fault localization technique on the 111 faulty versions is 0.1201. If we solely count the 55 faulty versions, which p-value is less than 1.0, the Pearson Correlation test of normality and P-score for the non-parametric fault-localization technique is 0.0037.

We may choose some reasonable threshold values (e.g., 0.700, 0.750, or higher) to determine whether these exist strong correlations between the degree of normality of the most fault-relevant predicate from some faulty version and the effectiveness of fault-localization techniques on this faulty version. However, the hypothesis $H3$ with $\theta_3 \geq 0.700$ can be always easily rejected. It means that, the answer to $Q8$ is *no*, and the normality for evaluation biases for predicates *weakly* correlates with the effectiveness of the non-parametric fault-localization technique. This can be also explained as that the non-parametric hypothesis test-



ing model for fault localization has high robustness in terms of the normality for the evaluation biases of predicates.

6.5.6 Threats to Validity

In this section, we discuss the threats to internal, construct, and external validity of our experiment.

Internal Validity

Internal validity is mainly caused by factors that affect experimental results.

The authors of SOBER have shared their instrumented faulty versions. We found that they missed some predicates. Since there is no evidence which of predicates should be included or not, we follow their specification to count all the predicates on and re-work the experiment. Therefore, the result of SOBER in our thesis is not the same as those reported previously.

We design null hypothesis and use hypothesis testing method to answer the interested research questions. However, to control the power of the designed null hypothesis, some parameters are involved in the research questions. For research questions *Q6*, *Q7*, and *Q8*, arbitrarily choosing these parameters does not have scientific support. To address this threat, we adopts some previously widely used value (e.g., 0.700 for correlation test), or changes the values in reasonable range to conduct hypothesis testing for several times with different thresholds.

In analyzing the scalability of our methods, CBI, and SOBER, we use the same programming library in implementation, and do not optimize any of them. Such consideration is to fairly compare their effectiveness. However, different implementation detail may affect their effectiveness and running time.

Many statistical algorithms are involved in our experiment, including the Mann-Whitney test, the Jarque-Bera test, and the Student's t-test. Different implementation details (such as accuracy in floating operations) may affect the experiment results. To address this threat, we choose using same programming library (ALGLib) to implement these algorithms. Such a consideration can reduce the implementation error and computing error.

Construct Validity

In the experiment, we include CBI and SOBER for comparison. However, there exist other techniques, and our intent is that both of CBI and SOBER are representative predicate-based techniques. We, therefore, adopt only CBI and SOBER as peer techniques and compare our method with them.



Another threat is the predicates we choose to investigate. Since CBI and SOBER reinterpret different kinds of statements as predicates, it seems not easy to compare them with each other directly. However, SOBER has reported that scalar-pair predicates have minor effects on fault-localization results. Hence, we follow SOBER to adopt the other two kinds of predicates in the experiment. The effect of including scalar-pair statements as predicates is not investigated in this thesis.

Although t-score is widely used in previous work (including [76][77]), some limitations have also been reported in the use of t-score (see [36] for example). Has any other measures been used to evaluate predicate-based techniques successfully? we are not aware of such alternatives in the public literature. Therefore, we involve a novel metrics, P-score. The P-score metrics evaluates the speed of locating the most fault-relevant predicate using the generated predicate list. The consideration is that all these techniques estimate the fault relevance of predicates and generate a list of predicates according to their suspiciousness.

We use the Mann-Whitney test, the Jarque-Bera test, the Student's t-test, and the Pearson Correlation test in the experiment. Using other kind of hypothesis testing method, normality test method, or correlation test method may produce different results. To address this issue, all the methods we choose are representative algorithms among their families.

Threats also exist in the manual work involved in the experiments, since we manually mark the most fault-relevant predicates. This step is neither entirely objective nor automated. However, for some faulty versions that are hard to identify the most fault-relevant predicate, we have excluded them from the experiment, and listed out these faulty versions. How to decide the most fault-relevant predicate for other programs can be discussed in future work.

External Validity

In this experiment, external validity is related to the target programs used.

We use a previously used metric, P-score to evaluate the effectiveness of fault-localization techniques. Using other kind of metrics may give different results. Since we cannot generate our results to other metrics, more future work may address this threat. For example, the metric t-score is also used in previous studies [76][77]. However, it is also reported to have some limitation [36].

In our experiment, we empirically show the results of using two non-parametric and two parametric hypothesis testing methods to locate fault. Using other kinds of non-parametric or parametric statistical methods may give different results.

External validity may also be caused by the target programs used. Since the



faults in Siemens programs are manually seeded, they may not truly represent realistic faults. Using other programs may give different results. On the other hand, the effect of using other metric is not discussed yet. More evaluation should, therefore, be done in the future.

Our experiment is conducted on the Siemens programs. Though the Siemens programs are representative programs and have been used in many previous studies [68][70][76][77], using other programs (e.g., JAVA programs instead of C programs) may product different results. Besides, using different normality test or hypothesis testing method may also affect the experiment results. More evaluation on using such alternatives should be included in future work.

6.6 Summary

Many previous studies contrast the feature spectra of passed and failed executions to locate the predicates correlated to faults (or to locate the faulty statements directly). However, they overlook the investigation of the statistical distributions of the spectra, on which their parametric techniques fully rely. In our previous work, we have argued and empirically verified that assuming a specific distribution of feature spectra of dynamic program statistics is problematic. It highlights a threat to construct validity in fault localization research that previous studies do not report in their empirical evaluation and model development.

In this chapter, we formally investigated the statistical nature about the normality test results for predicates. We showed empirically that the evaluation biases of predicates, and particularly the most fault-relevant predicates, are not necessarily normally distributed. In fact, almost all examined distributions of evaluation biases are either normal or far from normal, but not in between. Besides, the most fault-relevant predicates are less likely to exhibit normal distributions in their evaluation biases than other predicates. Our work highlights a threat to construct validity in fault-localization techniques that employ parametric hypothesis testing methods. We further investigated the effect of normality of predicates on fault-localization techniques, and used it as a measure to test the robustness of non-parametric fault-localization techniques. The empirical results showed that the non-parametric model for fault localization has high robustness in terms of the normality for the evaluation biases of predicates.

Our main contribution in this chapter is four-fold. (i) This chapter conducts the first experiment of using both standard non-parametric and parametric hypothesis testing methods in statistical fault localization. The empirical results enhance previous conclusion that using non-parametric hypothesis testing meth-



ods are effective for predicate-based fault-localization techniques. (ii) It is the first investigation on the normality nature of the execution spectra. The empirical results show that normal distribution is not common for the evaluation biases of predicates. In particular, the results indicate that the chance of the distribution of the evaluation biases of fault-relevant predicates being normal is less likely than that of other predicates. Such a finding highlights a threat to the construct validity of any empirical study which is based on the assumption that the evaluation biases of predicates form normal distributions. (iii) It proposes a new metrics P-score to measure the effectiveness of fault-localization techniques. (iv) It is the first study that investigates the effect of normality for the evaluation biases of predicates on non-parametric fault-localization techniques. The empirical results show that the effectiveness of our non-parametric fault-localization technique is weakly correlated to normality of the underlying distribution of evaluation biases.

Chapter 7

Further Discussion

In this chapter, we discuss several issues related to previous chapters.

7.1 Tie-breaking Strategy

For the statistical fault-localization techniques discussed in this thesis, the suspiciousness score of program elements may be identical to each other. In such a case, such program elements form a tie and they should be examined as a whole when evaluating the effectiveness of that technique.

In previous work, the tie-breaking strategy (see [96][112]) is often employed to optimize the baseline fault-localization techniques. For program elements in a tie case, their priorities are further determined using other formulas, such as the confidence formula $Conf(s_i)$ [112] used in the technique Tarantula. A tie-breaking strategy gives a better ranking list when one follows the ranking list to locate faults. On the other hand, a tie-breaking strategy does not modify the computed suspicious scores of the program entities by those techniques.

Our technique can also be optimized in exactly the same way. For example, the tie-break strategy can be applied on our techniques Slope, CP, or DES, to improve the effectiveness of them. Currently, we do not use any tie-breaking strategy in CP and DES. However, involving a tie-breaking strategy in CP or DES is believed to improve the effectiveness of them.

7.2 Adaptation of our Technique

Our techniques can be easily applied to other dimension. For example, the heuristics of Slope can be also used in function-level, module-level, branch-level, and so on. Since such work can be developed by direct applying the thought of Slope, we do not list their results in this thesis.



Take our technique CP as an example. We capture the propagation of infected program states via CFG. In fact, other flow-graph representations of program executions, such as program dependency graphs [5] or data flow graphs, may be employed to replace CFG. We do not iteratively show how to adapt each of them in our technique.

7.3 Fault Fix after Fault Localization

In a complete software debugging process, programmers need to fix the fault after locating it. However, this is out of the scope of this thesis work.

7.4 Oracle Problem before Fault Localization

Software testing is a key activity in any software development project. It assures applications by executing programs over test cases with the intent to reveal failures [9]. Definitely, statistical fault localization relies on the output of software testing, that is, the pass/fail status of those test cases.

In this section, we first give necessary background on oracle problem in testing, and then design research questions to understand the properties of two testing methods, Metamorphic Testing and Assertion Checking. In the experiment section, we report the empirical results and analyze the empirical results.

This section is based on our previous work [118].

7.4.1 Background

To conduct testing, software testers usually evaluate the test results through an oracle, which is a mechanism for checking whether a program behaves correctly [108]. Many programs do not have a full specification, and many of them are developed without similar versions for reference. In these situations, oracles may be unavailable or too expensive to apply. This is known as the test oracle problem [108]. The oracle problem is not limited to the above kinds of scenarios. For instance, for programs involving complex computations (such as partial differential equations [31], graphics-based software [21][24][25], database applications [101], large-scale components, web server, or operating systems [101]), their outputs are difficult to verify. In current software practices, the oracle is often a human tester who checks the testing results manually. The manual checking of program output acutely limits the efficiency of testing and increases its cost, especially when there is a need to verify the results of a large number of test cases. Assessing the correctness of



program outcomes has, therefore, been recognized as “one of the most difficult tasks in software testing” [79].

As we have discussed in Section 7.4.1, metamorphic testing relies on a checking condition that relates multiple test cases and their results in order to reveal failures. Such a checking condition is known as a *metamorphic relation*. In this section, we revisit metamorphic relations and discuss how they can be used in the metamorphic approach to software testing.

Metamorphic Relations

A *metamorphic relation* (MR) is a relation over a series of distinct inputs and their corresponding results for multiple evaluations of a target function [2]. Consider, for instance, the sine function. We have the following relation: *If $x_2 = \pi - x_1$, then $\sin x_2 = -\sin x_1$.* We note from this example that a metamorphic relation consists of two parts. The first part (denoted by r in the definition below) relates x_2 to x_1 . The second part (denoted by r') relates the results of the function. If the MR above is not satisfied for some input, we deem that a failure is revealed.

Definition 7.2.1 (Metamorphic Relation) [21] *Let $\langle x_1, x_2, \dots, x_k \rangle$ be a series of inputs to a function f , where $k \geq 1$, and let $\langle f(x_1), f(x_2), \dots, f(x_k) \rangle$ be the corresponding series of results. Suppose $\langle f(x_{i1}), f(x_{i2}), \dots, f(x_{im}) \rangle$ is a subseries, possibly an empty subseries, of $\langle f(x_1), f(x_2), \dots, f(x_k) \rangle$.² Let $\langle x_{k+1}, x_{k+2}, \dots, x_n \rangle$ be another series of inputs to f , where $n \geq k + 1$, and let $\langle f(x_{k+1}), f(x_{k+2}), \dots, f(x_n) \rangle$ be the corresponding series of results. Suppose, further, that there exists relations $r(x_1, x_2, \dots, x_k, f(x_{i1}), f(x_{i2}), \dots, f(x_{im}), x_{k+1}, x_{k+2}, \dots, x_n)$ and $r'(x_1, x_2, \dots, x_n, f(x_1), f(x_2), \dots, f(x_n))$ such that r' must be true whenever r is satisfied. Here, r and r' can be any mathematics relation of aforementioned parameters. We say that*

$$\begin{aligned} \mathbf{MR} = & \{ \langle x_1, x_2, \dots, x_n, f(x_1), f(x_2), \dots, f(x_n) \rangle \\ & | r(x_1, x_2, \dots, x_k, f(x_{i1}), f(x_{i2}), \dots, f(x_{im}), x_{k+1}, x_{k+2}, \dots, x_n) \\ & \rightarrow r'(x_1, x_2, \dots, x_n, f(x_1), f(x_2), \dots, f(x_n)) \} \end{aligned}$$

is a metamorphic relation. When there is no ambiguity, we simply write the metamorphic relation as

$$\begin{aligned} \mathbf{MR}: & \text{If } r(x_1, x_2, \dots, x_k, f(x_{i1}), f(x_{i2}), \dots, f(x_{im}), x_{k+1}, x_{k+2}, \dots, x_n) \\ & \text{then } r'(x_1, x_2, \dots, x_n, f(x_1), f(x_2), \dots, f(x_n)). \end{aligned}$$

² $\langle x_1, x_2, \dots, x_k \rangle$ and $\langle f(x_1), f(x_2), \dots, f(x_k) \rangle$ denotes a series of inputs and its corresponding series of results.



Furthermore, x_1, x_2, \dots, x_k are known as *initial test cases* and $x_{k+1}, x_{k+2}, \dots, x_n$ are known as *follow-up test cases*.

Similar to assertions in the mathematical sense, metamorphic relations are also necessary properties of the function to be implemented. They can, therefore, be used to detect inconsistencies in a program. They can be any relations involving the inputs and outputs of *two or more* executions of the target program. They may include inequalities, periodicity properties, convergence properties, subsumption relationships, and other properties.

Intuitively, human testers are needed to study the problem domain related to a target program and formulate metamorphic relations accordingly. This is akin to requirements engineering, in which humans instead of automatic requirements engines are necessary for formulating systems requirements. In some domains where the requirements of an implementation are best specified mathematically, metamorphic relations may readily be identified. Is there a systematic methodology guiding testers to formulate metamorphic relations like the methodologies that guide systems analysts to specify requirements? This remains a challenging question. We shall further investigate along this line in the future. We observe that other researchers are also beginning to formulate important properties in the form of specifications to facilitate the verification of system behaviors [37].

Metamorphic Testing

In practice, if the program is written by a competent programmer, most test cases will be passed test cases. These passed test cases have been considered useless in conventional testing. Metamorphic testing (MT) uses information from such passed test cases, which will be referred to as *initial test cases*.

Consider a program p for a target function f in the input domain D . A series of initial test cases $T = \langle t_1, t_2, \dots, t_k \rangle$ can be selected according to any test case selection strategy. Executing the program p on T produces outputs $p(t_1), p(t_2), \dots, p(t_k)$. When there is a test oracle, the test results can be verified against $f(t_1), f(t_2), \dots, f(t_k)$. If these results reveal any failure, testing stops. On the other hand, when there is no test oracle or when no failure is revealed, the metamorphic testing procedure can continue to be applied to automatically generate follow-up test cases $T' = \{t_{k+1}, t_{k+2}, \dots, t_n\}$ based on the initial test cases T so that the program can be verified against metamorphic relations.

Definition 7.2.2 (Metamorphic Testing) [21] *Let P be an implementation of a target function f . The metamorphic testing of the metamorphic relation*

MR: *If $r(x_1, x_2, \dots, x_k, f(x_{i1}), f(x_{i2}), \dots, f(x_{im}), x_{k+1}, x_{k+2}, \dots, x_n)$,*



then $r' (x_1, x_2, \dots, x_n, f(x_1), f(x_2), \dots, f(x_n))$

involves the following steps: (i) Given a series of initial test cases $\langle x_1, x_2, \dots, x_k \rangle$ and their respective results $\langle P(x_1), P(x_2), \dots, P(x_k) \rangle$, generate a series of follow-up test cases $\langle x_{k+1}, x_{k+2}, \dots, x_n \rangle$ according to the relation $r (x_1, x_2, \dots, x_k, P(x_{i1}), P(x_{i2}), \dots, P(x_{im}), x_{k+1}, x_{k+2}, \dots, x_n)$ over the implementation P . (ii) Check the relation $r' (x_1, x_2, \dots, x_n, P(x_1), P(x_2), \dots, P(x_n))$ over P . If r' is false, then the metamorphic testing of MR reveals a failure.

Metamorphic Testing Procedure

Gotlieb and Botella [54] developed an automated framework for a class of metamorphic relations. The framework translates a specification into a constraint logic programming (CLP) program. Test cases can be automatically generated according to the CLP program using a constraint solving approach. Their framework works on a subset of the C language, but it is not clear whether the framework is applicable to test cases involving objects. Since we want to apply MT to object-oriented programs, we adopt the original procedure [19], which is described as follows:

First, testers identify and formulate metamorphic relations MR_1, MR_2, \dots, MR_n from the target function f . For each metamorphic relation MR_i , testers construct a function gen_i to generate follow-up test cases from the initial test cases. Next, for each metamorphic relation MR_i , testers construct a function ver_i , which will be used to verify whether multiple inputs and the corresponding outputs satisfy MR_i . After that, testers generate a set of initial test cases T according to a preferred test case selection strategy. Finally, for every test case in T , the test driver invokes the function gen_i to generate follow-up test cases and apply the function ver_i to check whether the test cases satisfy the given metamorphic relation MR_i . If a metamorphic relation MR_i is violated by any test case, ver_i reports that an error is found in the program under test.

7.4.2 Research Questions

Like other comparisons of testing strategies such as comparison between control flow and data flow test adequacy criteria [37] and comparison among different data flow test adequacy criteria [60], controlled experimental evaluations are essential. They should answer the following research questions.

- (a) *Can the subjects properly apply MT after training?* Can the subjects identify correct and useful metamorphic relations from target programs? Can the same metamorphic relations be discovered by



multiple subjects?

- (b) *Is MT an effective testing method?* Does MT have a comparative advantage over other testing strategies such as assertion checking in terms of the number of mutants detected? To address this question, we shall use the standard statistical technique of null hypothesis testing.

Null Hypothesis H_0 : There is no significant difference between MT and assertion checking in terms of the number of mutants detected.

Alternative Hypothesis H_1 : There is a significant difference between MT and assertion checking in terms of the number of mutants detected.

We aim at applying the standard concept of the p-value in the Mann-Whitney test to find the confidence level that H_0 can be rejected, with a view to supporting our claim that the difference between MT and assertion checking is statistically significant rather than by chance.

- (c) What is the effort, in terms of time cost, in applying MT?
- (d) If an MR is faulty, what is the cost (in terms of the number of mutants detected) of applying it?

7.4.3 Experiment

This section describes the set up of the controlled experiment. It first formulates the research questions to be investigated and then describes the experimental design and experimental procedure.

Our experiment identifies four independent and three dependent variables. The independent variables are *testing strategies*, *subjects*, *target programs*, *faulty versions of target programs*, and *faulty versions of metamorphic relation programs*. The dependent variables are *time cost*, *number of metamorphic relations/assertions*, and *testing effectiveness in terms of mutation detection ratio*. For the variable on testing strategies, we incorporate MT and assertion checking. In the rest of this section, we describe the other three independent variables. Later sections will analyze the results according to the dependent variables.

Target Programs

We used three open-source programs as target programs. All of them were Java programs selected from real-world software systems.



Table 7.1: Statistics of target programs

Program	Number of LOC	Number of Methods	Number of Output Affecting Methods
Boyer	241	16	9
BooleanExpression	231	15	12
TxnTableSorter	281	18	15

The first target program `Boyer` is a program using the Boyer-Moore algorithm to support the applications in Canadian Mind Products, an online commercial software company (available at <http://mindprod.com/products1.html>). The program returns the index of the first occurrence of a specified pattern within a given text.

The second target program `BooleanExpression` evaluates Boolean expressions and returns the resulting Boolean values. For example, the program may evaluate the expression “!(true && false) || true” and returns *true*. The program is a core part of a popular open-source project `jboolexp` (available at <http://sourceforge.net/projects/jboolexp>) in SourceForge (URL <http://www.sourceforge.net>), the largest open-source project website.

The third target program is `TxnTableSorter`. It is taken from a popular open-source project `Eurobudget` (<http://eurobudget.sourceforge.net>) in the SourceForge website. Eurobudget is an office application written in Java, similar to Microsoft Money or Quicken.

Table 7.1 shows the statistics of the three target programs. The first program is a piece of commercial software. The second program is a core part of a standard library. The third one is selected from real office software with hundreds of classes and more than 100,000 lines of code in total. All of them are open source. The sizes of these programs are in line with the sizes of target programs used in typical software testing researches such as [2], in which it uses the Siemens suites.

Faulty Versions of Target Programs

To investigate the relative effectiveness of metamorphic testing and assertion checking, we use mutation operators [59] to seed faults to programs. A previous study [2] shows that a set of well-defined mutation operators can simulate the real environment for testing experiments.

In our experiment, mutants are seeded using the tool `muJava` [80]. The tool supports two levels of mutation operators: class level and method level. Class level mutation operators are operators specific to generating faults in object-oriented programs at the class level. Method level mutation operators defined in work [87] are operators specific for statement faults. We only seed method



Table 7.2: Categories of mutation operators

Category	Description
AOD	Delete Arithmetic Operator
AOI	Insert Arithmetic Operator
AOR	Replace Arithmetic Operator
ROR	Replace Relational Operator
COR	Replace Conditional Operators
COI	Insert Conditional Operator
COD	Delete Conditional Operator
SOR	Replace Shift Operator
LOR	Replace Logical Operator
LOI	Insert Logical Operator
LOD	Delete Logical Operator
ASR	Replace Assignment Operators

level mutation operators to the programs under study because our experiment focuses on unit testing and this set of operators has been studied extensively in the software engineering research community [2][17][26][53][86][87]. Table 7.2 lists all the mutation operators used in the controlled experiment.

General speaking, muJava examines each statement in a given program and then applies each applicable mutation operator to generate a variant of the program. In other words, for each statement and each applicable mutation operator, it produces a *single-fault* version of the given program. It has been well-recognized in the software engineering research community that single-fault mutants couple well with high-order mutants and real faults and using them to conduct test experiment can adequately simulate realism [2][87]. On the other hand, research on finding an adequate subset of mutation operators to replace the entire set is still going on [86]. Many software engineering researchers continue to use the full set of mutants constructed from a tool to conduct experiments.

A total of 151 mutants are generated by muJava for the class Boyer, 145 for the class BooleanExpression, and 378 for TxnTableSorter. Note that faults are only seeded into the methods supposedly covered by the test cases for unit testing. Table 7.3 lists the number of mutants under each category of operators. We create a faulty version for each mutant. Finally, we used all the $151 + 145 + 378 = 674$ single-fault versions in the controlled experiment.



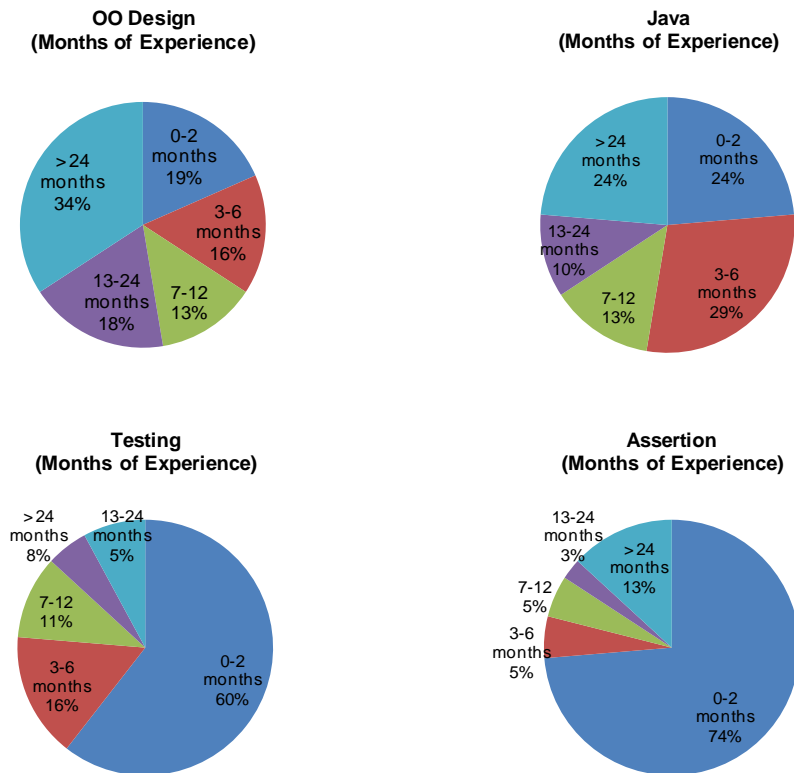


Figure 7.1: Experiences of subjects in object-oriented design, Java, testing, and assertions

Controlled Variables

All the 38 subjects are graduate students in computer science or equivalent who attend the course “Advanced Topics in Software Engineering: Software Testing” at The University of Hong Kong. These students have at least a bachelor degree in computer science, computer engineering, or electronic engineering. The majority of them are part-time MSc students with some industrial experience. The rests are MPhil and PhD students. We control that the training sessions of either approach are comparable in duration and in content. The number of subjects used our controlled experiment is similar to

Table 7.3: Number of single-fault programs by mutation operator category

Program	AOD	AOI	AOR	COD	LOI	ROR	LOR	COR	COI	ASR	Total
Boyer	1	85	14	0	24	16	3	2	1	5	151
BooleanExpression	3	86	3	1	22	27	0	3	0	0	145
TxnTableSorter	8	226	16	0	71	43	2	7	5	0	378

those in other software engineering controlled experiments. For instance, the experiments in [59][105] use 44 subjects.

Since differences in software engineering background might affect the students' capability to apply metamorphic testing or assertion checking, we conduct a brief survey prior to the experimentation. The survey asks subjects their experiences in the industrial environment in each of the following four areas: object-oriented design, Java programming, software testing, and assertion checking.

Figure 7.1 lists the result of the survey. The overall survey result shows that most of them have real-life or academic experience. As most of subjects are knowledgeable about object-oriented design and Java programming, they are deemed to be competent in the tasks in the controlled experiment. On the other hand, we find a few students having rather limited experience in software testing and assertion checking. Since they have no prior concepts of metamorphic testing either, the experiment does not specifically favor the metamorphic approach.

Experimental Setup

Before the experiment, the subjects are given a six-hour training to use MT and assertion checking. We carefully monitor the time durations so that the times allocated to train either technique are roughly equal to each other. (We could not have identical durations for both techniques; otherwise, the same testing background such as the concept of test oracles in general would needlessly be introduced twice to the subjects.) The target programs and the tasks to be performed are also presented to the subjects. The subjects are briefed about the main functionality of each target program and the algorithm used, thus simulating the process in real-life in which a tester acquires the background knowledge of the program under test. They are blind to the use of any mutants in the controlled experiment. For each program, the subjects are required to apply MT strictly following the procedure described in previous section, as well as to add assertions to the source code for checking. We do not restrict the number of metamorphic relations and assertions to be associated to individual target programs. The subjects are told to develop metamorphic



relations and assertions as they consider suitable, with a view to thoroughly test each target program.

We do not mandate the use of a particular testing case generation strategy, such as all-def-use criterion or random testing or specification-based approach, for either MT or assertion checking. The subjects are simply asked to provide adequate test cases for testing the target programs. This avoids the possibility that some particular test case selection strategy, when applied in large scale, might favor either MT or assertion checking.

We ask the students to submit metamorphic relations, functions to generate follow-up test cases, functions to verify metamorphic relations, test cases for metamorphic testing, source code with inserted assertions, and test cases for assertion checking. They are also asked to report the time costs to apply metamorphic testing and assertion checking. Before testing the faulty versions with these functions, assertions, and test cases, we check their submissions carefully to ensure that there is no implementation error.

Results and Analysis

This section presents the experimental results of applying metamorphic testing and assertion checking. They are structured according to the dependent variables presented in the last section.

Feasibility of MR Development and Assertion Development

A critical and difficult step in applying MT and assertion checking is to develop metamorphic relations and assertions for the target programs. Table 7.4 reports the number of metamorphic relations and assertions identified by the subjects for the three target programs. The mean numbers of metamorphic relations developed by the subjects for the three programs are 2.79, 2.68, and 5.00, respectively. The total numbers of distinct metamorphic relations identified by all subjects for the three programs are 18, 39, and 25, respectively. The mean numbers of assertions for the three programs are 6.96, 11.35, and 10.97, respectively.

First, we observe that all the subjects can properly create metamorphic relations and assertions after training. We further inspect their metamorphic relations and assertions, and find that many of the identified artifacts overlap among subjects. Take Boyer as an example. There are 38 subjects in total. They collectively identify 18 distinct metamorphic relations, and on average, each subject identifies 2.79 metamorphic relations. In other words, if all the metamorphic relations identified are distinct, there should be 108 metamorphic relations. It means that, on average, each distinct metamorphic relation is



Table 7.4: Number of metamorphic relations and assertions

Program	Total	No. of Metamorphic Relation				No. of Assertions			
		Mean	Max	Min	StdDev	Mean	Max	Min	StdDev
Boyer	18	2.79	5	1	1.66	6.96	43	1	8.94
BooleanExpression	39	5.00	12	1	3.01	11.35	49	1	9.69
TxnTableSorter	25	2.68	7	1	1.59	10.97	36	2	10.97

discovered by six subjects (or 15.7% of the population). We also observe a similar result for assertion checking. This result is encouraging. It indicates that the identification of metamorphic relations can be practical and may share among different developers. It further answers another important research question on whether the same metamorphic relation can be discovered by more than one subject. The answer is “yes”.

To observe the variations in the feasibility of discovering metamorphic relations and assertions, we further normalize the standard derivations against the corresponding mean values in Table 7.4 for each program. The results are shown in Table 7.5. We observe that the standard deviations for discovering metamorphic relations are much larger than those for discovering assertions. In addition, we observe that the normalized standard deviations for discovering metamorphic relations across the three programs are quite consistent (close to 0.60 in each case). On the other hand, for assertions, the standard deviations trends vary from 0.20 to 0.30, which indicate a relatively larger fluctuation among programs. This initial finding may indicate that discovering metamorphic relations can be less dependent on the type of program being studied than discovering assertions. In other words, it suggests that there may be some hidden dominant factors (independent of the nature of target programs) governing the discovery of metamorphic relations. It will be interesting to identify these factors in the future.

On the other hand, we observe from Table 7.5 that the absolute values of the normalized standard deviations for discovering assertions are much smaller than those of metamorphic relations. It shows that our subjects

Table 7.5: Normalized standard derivations

Program	Metamorphic Relation	Assertion Checking
Boyer	0.59	0.21
BooleanExpression	0.60	0.20
TxnTableSorter	0.59	0.30



produce more predictable number of assertions. It may give project managers good guidelines to allocate project resources if they assign their programmers to do assertion checking in their software applications.

Size and Granularity of Metamorphic Relations and Assertions per Program

In general, the subjects can identify a larger number of assertions than metamorphic relations. As shown in Table 7.4, the maximum number of metamorphic relations discovered by subjects is almost the same as the mean number of assertions discovered by subjects. This suffices to indicate that there is a significant difference between the numbers of artifacts produced by the two testing methods.

We also observe that the subjects' abilities to identify metamorphic relations and assertions vary. This is understandable and agrees with the intuition that different developers may have quite diverse programming abilities. Take BooleanExpression as an example. Some subjects can identify 12 metamorphic relations and 49 assertions, while some others can only identify one metamorphic relation and one assertion.

We further observe from Table 7.4 that, for the three target programs, the ratios of the mean number of identified metamorphic relations to the mean number of identified assertions are 0.40, 0.44, and 0.24, respectively. If the effectiveness between the use of metamorphic testing and the use of assertion checking to identify failures is comparable, these ratios indicate that metamorphic relations can achieve a more coarse-grained granularity than assertions does. If so, we believe that MT helps developers raise the level of abstraction more than assertion checking does. Our data analysis to be presented in the next section will validate whether the effectiveness of the two methods are comparable.

Comparison on Fault Detection Capabilities

We use the subjects' metamorphic relations, assertions, and source and follow-up test cases to test the faulty versions of the target programs. The mutation detection ratio [2][59][87] is used to compare the fault detection capabilities of MT and assertion checking strategies. The *mutation detection ratio* of a test set is defined as the number of mutants detected by the test set over the total number of mutants [59]. For metamorphic testing, a mutant is detected if a source test case and follow up test cases executed on the mutant do not satisfy some metamorphic relations. For assertion checking, a mutant is detected if a



mutated statement is executed by a test case to enter an erroneous state that triggers an assertion statement.

For the sake of fairness, we apply these two methods to the *same* set of test cases separately. The source and follow-up test cases from metamorphic testing are both applied to assertion checking.

The average sizes of the test suites (including source and follow-up test cases) used by all students for the three programs were 19.9, 22.2, and 16.8, respectively. We also analyzed all the mutants manually before testing, and remove the equivalent mutants. There are 19, 18, and 61 equivalent mutants for program Boyer, BooleanExpression, and TxnTableSorter, respectively. We do not include them when calculating mutation detection ratios as these mutants cannot be detected by any test cases.

Table 7.6 reports on the mutation detection ratios for each program using the two testing methods. It shows that the mutation detection ratios by applying MT ranged from 44% to 93% for program Boyer, from 46% to 89% for program BooleanExpression, and from 32% to 74% for program TxnTableSorter.

Under the “Aggregate” columns are the percentages of mutants detected by all subjects. For MT, the mutation detection ratios were 98%, 95%, and 83%, respectively. Each entry was *significantly* better than the corresponding mutation detection ratio for assertion checking. This result, again, is encouraging.

The p-value of the standard Mann-Whitney test was less than 0.001 in all cases. Hence, we reject the null hypothesis H_0 on the effectiveness of fault detection at a 99.9% confidence level. In other words, MT may not only be comparable to assertion checking, but outperforms the latter. We use the same

Table 7.6: Mutation detection ratios for metamorphic testing and assertion checking

Program	Metamorphic Testing					Assertion Checking					Result of Mann-Whitney Test
	Mean	Max	Min	Stdev	Aggregate	Mean	Max	Min	Stdev	Aggregate	
Boyer	60%	93%	44%	0.13	98%	40%	66%	27%	0.12	81%	< 0.001
Boolean-Expression	63%	89%	46%	0.11	95%	39%	66%	30%	0.10	78%	< 0.001
Txn-TableSorter	59%	74%	32%	0.14	83%	37%	58%	22%	0.11	63%	< 0.001



Table 7.7: Statistics of time costs for applying MT and assertion checking

		Smallest Observation	Lower Quartile	Lower Notch	Median	Upper Notch	Upper Quartile	Largest Observation
Boyer	MT	0.58	1.73	1.99	2.51	5.01	5.11	9.82
	Assertion	0.58	1.03	1.03	1.48	1.99	2.12	2.18
Boolean-Expression	MT	0.32	2.25	2.51	3.28	6.03	8.02	12.71
	Assertion	0.45	1.35	1.48	1.99	3.02	5.01	7.77
Txn-TableSorter	MT	0.26	2.51	3.02	3.98	6.03	6.99	11.68
	Assertion	0.52	1.03	1.28	1.99	3.02	3.98	6.74

set of test cases when applying the Mann-Whitney test.

These setting and hypothesis testing results indicate that the difference is attributed by the ability to *violate* the constraints specified via metamorphic relations and those specified via assertion checking. We observe that the difference between the two testing methods in our experiment is whether the constraint is specified for one execution or for multiple executions. The former type of constraint is for assertion checking, and the latter type is for metamorphic relation. In the other words, the result indicates that using the test results of multiple executions to identify failures collectively is more effective than just using one execution.

Although our empirical results show that metamorphic testing can be effective, there is a need to develop systematic methods for creating metamorphic relations and assertions (because individual tester's results were lower than the aggregated results of all testers in either approach). The average differences between the mean column and the aggregate column for MT and assertion checking were 41.3% and 35.3%, respectively. The standard derivations did not differ much statistically. They ranged from 0.10 to 0.14, as shown in Table 7.6.

Comparison of Time Cost

We would like to compare the time costs between metamorphic testing and assertion checking. From the subjects' submissions, we found that they spent less time on applying assertion checking than metamorphic testing.

Table 7.7 shows the statistics of the time costs for applying the respective strategies to the target programs. Each entry in the column "Smallest Observation" stands for the smallest value (time cost in terms of hours) in the respective data set. Each entry in the column "Largest Observation" stands for the largest value in the respective data set. Each entry under "Median" captures the 50th percentile in the data. The entries under "Lower Quartile" and "Upper Quartile" capture the values of the 25th and 75th percentiles (in



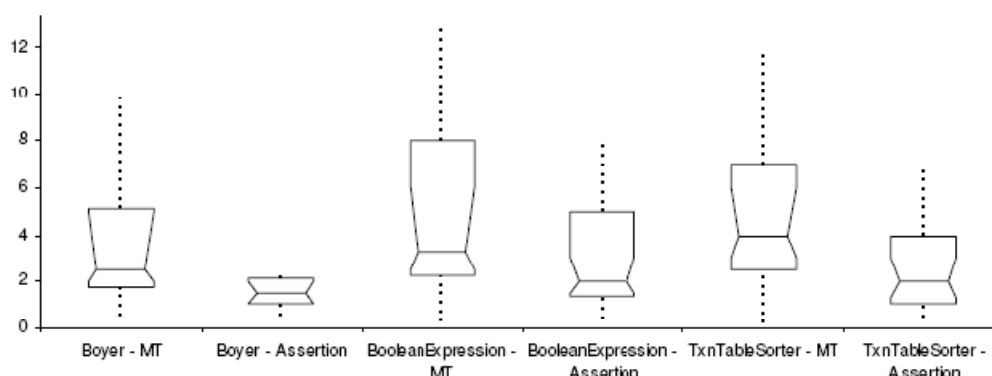


Figure 7.2: Box-and-whisker plots of time costs for applying MT and assertion checking

the order from small to large) in the data, respectively. The entries under “Lower Notch” and “Upper Notch” display the variability of the median in the data set.

We observe from Table 7.7 several interesting tradeoffs between MT and assertion checking. First, the smallest observation in assertion checking is consistently larger than that in MT. Although applying MT is apparently more complex than assertion checking, this result shows that, for the most effective testers, the effort to design and implement metamorphic relations is less than the effort to design and implement assertions. Second, for each of the three target programs, the median and the largest observation in MT are always greater than the corresponding values in assertion checking. It indicates that designing and implementing MRs is generally more time-consuming than designing and implementing assertions. Third, from the lower quartiles and upper quartiles in MT and assertion checking for these programs, we further observe that the time spent on MT varies more drastically than the time on assertion checking.

Intuitively, many developers have developed skills to understand program logic from source code and are comfortable in conducting program comprehension. Furthermore, developers are used to modifying an existing program to implement new changes to the source code. In view of the above intuition, we believe that adding assertions to source code is a more familiar and handy task for the subjects than formulating and implementing MRs.

To analyze the differences between these two testing approaches to alleviate the test oracle problem, we further represent their time costs using box-and-whisker plots. Figure 7.2 shows the plots for applying the respective strategies to the target programs. The time cost for MT includes the time to identify and formulate test cases, write functions to generate follow-up test

Table 7.8: Mutation detection ratios for MT
with and without faulty metamorphic relation implementations

Program	With Correct MR Implementations Only					With Both Correct and Faulty MR Implementations				
	Mean	Max	Min	Stdev	Median	Mean	Max	Min	Stdev	Median
Boyer	59%	100%	2%	0.25	56%	95%	100%	85%	0.03	95%
Boolean-Expression	72%	100%	34%	0.27	85%	91%	100%	80%	0.08	94%
Txn-TableSorter	66%	100%	6%	0.22	57%	91%	100%	67%	0.05	90%

cases, and write functions to verify the identified metamorphic relations. The time cost for assertion checking includes the time spent on adding assertions to the source code.

The vertical axis of Figure 7.2 shows the time cost in number of hours. The bottom and top horizontal lines of each box indicate the lower and upper quartiles. The whiskers, drawn as dotted vertical lines, show the full range of the data. The median is drawn as a horizontal line inside each box. A notch is added to each box to show the uncertainty interval for each median. If two median notches do not overlap, it indicates that there is a statistically significant difference between the two medians at a 95% confidence level.

For Boyer and TxnTableSorter, there is a significant difference between the times spent in applying metamorphic testing and assertion checking. The difference is less statistically significant for BooleanExpression. The exact values of the respective notches can be found in Table 7.7.

The difference in time cost is acceptable for a number of reasons. First, the time costs for MT implementations include the generation of follow-up test cases, whereas the time costs for assertion checking do not include the generation of any test cases. Second, some subjects have had prior experience in assertion checking. We believe that the extra time spent on developing programs to generate follow-up test cases have paid off because, as discussed in later section, these (follow-up) test cases have demonstrated to be very useful in detecting failures of the target programs. Furthermore, although there are statistically significant differences in time costs (especially if we view Table 7.7 in relative terms), we also note that the actual median difference in absolute terms ranges between one to two hours in the experiment.

Figure 7.2 further indicates that the time cost for applying MT to object-oriented testing at the class level is acceptable compared to that of assertion checking. When we consolidate the comparisons in later sections, we find that MT provides a stronger oracle check with a tradeoff of slightly more time for preparation.



Comparison of MT with and without Faulty MR Implementations

As we have highlighted, an MR is a property that the correct version of a program under test should exhibit. To apply MT automatically, testers need to execute the implementations of the MRs for the program under test. In the controlled experiment, these MR implementations are constructed by the subjects. It is crucial to know whether MT can still be effective if MR implementations can be faulty.

We thus conduct a follow-up experiment to validate whether MT is robust enough if faulty metamorphic relations are used to detect failures in the subject programs. We use the set of mutation operators of muJava mentioned above to generate single-fault mutants of the MR implementations. In total, muJava produces 88, 71, and 89 MR mutants for the three subject programs, respectively. If an MR mutant cannot be killed by any test case, we exclude it from the follow-up experiment. We also exclude similar target program mutants. We then select a test suite of 20 test cases randomly from the test pool for each target program and compute the mutation detection ratio accordingly. We note that, in this validation experiment, a revealed failure may be a mistake (namely, a false positive case) produced by a faulty MR implementation, a failure of the faulty target program, or both. We repeated the experiment by selecting the test suites 10 times.

The result is shown in Table 7.8. First, if we only use correct MRs to identify failures, the mean fault detection rate in the validation experiment is close to the mutation detection rate shown in Table 7.6. It indicates that the results of the validation experiment are comparable to the above-mentioned experiment that compares MT and assertion checking. Second, if MR implementations can be faulty, the mean value is much higher (consistently over 90% as shown in the rightmost column of Table 7.8). The result indicates that a test suite is likely to detect problems in the combination of a faulty target program and a set of faulty MRs. This finding is encouraging because MT can still be reasonably applied even if some MR implementations may be faulty. If a faulty MR implementation can be debugged successfully, we believe that the failure detection rate of the test suite will drop, as indicated by the comparison in Table 7.7. However, fixing the faults in the MR implementations will incur additional time cost. It may make the difference in time cost between metamorphic testing and assertion checking more noticeable. Thus, it warrants more study to find the extent that testers should stop further maintenance of a faulty MR implementation in order to balance the development cost and product quality.

Further Discussions on MT



Table 7.9: Examples of Metamorphic Relations for program Boyer

Index	Metamorphic Relation
MR ₁	If $(x_1 = \text{concatenate}(x_2, x_3)) \wedge (\text{find}(x_2, x_4) > -1)$, then $\text{find}(x_1, x_4) = \text{find}(x_2, x_4)$.
MR ₂	If $(x_1 = \text{concatenate}(x_2, x_3)) \wedge (\text{find}(x_2, x_4) = -1) \wedge (\text{find}(x_3, x_4) > -1)$, then $\text{find}(x_1, x_4) \leq \text{length}(x_2) + \text{find}(x_3, x_4)$.
MR ₃	If $(x_1 = \text{concatenate}(x_2, x_3)) \wedge (\text{find}(x_1, x_4) = \text{length}(x_2))$, then $\text{find}(x_3, x_4) = 0$.

The function *concatenate* (x, y) returns the result of concatenating string x and string y . The function *find* (x, y) returns the zero-based index of string y within the string x if x contains y ; otherwise, it returns -1 .

In general, we observe that the more MRs being used, the higher will be the mutation detection ratio. As we have indicated, there is a need to propose more systematic methods to construct the implementation of metamorphic relations. The utilization of an MR implementation also increases as testers increase the number of initial test cases applicable to the MR. Since the resources in software testing are often limited, it is also worth investigating the number of test cases adequate for MT.

Moreover, testers may apply a number of metamorphic relations in order to test a program. In general, different metamorphic relations have non-identical fault detection capabilities. Let us, for instance, analyze the experimental results of the Boyer program. The subjects have identified 18 metamorphic relations in total. We observe that four subjects have only identified one and the same metamorphic relation (MR₁ in Table 7.9). The implementation of this metamorphic relation constructs a follow-up test case by appending an arbitrary string to the string in the initial test case and reusing the given pattern in the initial test case. It also checks whether the Boyer program over the two test cases will give the same outputs if the program locates successfully the given pattern in the initial string. The mutation detection ratios resulting from these MR implementations by the subjects are no more than 60% no matter how many test cases they used. We also find that some subjects using the other metamorphic relations (MR₂ and MR₃ in Table 9) achieve mutation detection ratios higher than 80%, although they only propose four initial test cases. It indicates that the quality of metamorphic relations can be a key factor in determining the effectiveness of MT.

Threats to Validity

We describe the threats to validity in the following sections.



Internal Validity

For this experiment, we provide the subjects with all the background materials and confirm with them that they have sufficient time to perform all the tasks. On the other hand, we appreciate that students might be interrupted by minor Internet activities when they perform their tasks. Hence, the time costs reported by the subjects should be viewed and analyzed conservatively. Furthermore, the subjects do not know the nature and details of the faults seeded. This measure ensures that their “designed” metamorphic relations and assertions are unbiased with respect to the seeded faults.

We use test cases provided by our subjects to conduct the experiment. We do not know whether these test cases may favor assertion checking, metamorphic testing, or neither of them. We do not disclose the purpose of the experiment to any subjects, and only request them to produce test cases that they consider sufficient for both metamorphic testing and assertion checking. To address the threat to internal validity, we use all test cases from different subjects on every applicable MR. Since subjects do not communicate with one another in the experiment, this setting helps disassociate test cases from particular MRs.

Readers may be concerned whether the target programs can be faulty. We have carefully checked the classes before the experiment. Furthermore, none of the subjects has reported any errors in the target programs. Another concern is whether the developed MRs may contain faults. To address this threat, we have run all test cases by all subjects as well as our own test cases on all these MRs for the target programs. We observe no failure in the verification exercise. To further address this risk, we have also conducted a verification experiment to explicitly test the mutants of the implementations of the metamorphic relations.

External Validity

In this experiment, external validity is related to the degree to which the results can be generalized to the testing of real-world systems. The programs used in our experiment are from real-life applications. For example, Eurobudget is widely used and has been downloaded more than 10,000 times from SourceForge. On the other hand, some real-world programs can be much larger and less well documented than the open-source programs studied. More future studies may be in order for the testing of large complex systems using the MT method. We use the MR implementations produced by our subjects. Other testers of other target programs may produce other MR implementations. Additional experiments should always be helpful in



improving the generalization of the results that we obtain and present in this thesis.

We use Java programs in the experiments, and all MR implementations are naturally written in Java. Although Java programs are widely used in practice, an MR is inherently a property. It may also be intuitive to implement an MR using a rule-based approach via logic programming. It is not immediately obvious to us whether the use of a rule-based approach may produce different comparison results.

We use the test cases produced by the subjects. The use of other schemes (such as statement coverage) may produce different sets of test cases.

Construct Validity

We measure the effectiveness of metamorphic testing and assertion checking via a mutation detection ratio. Mutation analysis has been used and validated to be reliable for testing experiments that stimulate real fault scenarios for deterministic, procedural programs (written in C) [2]. The use of mutation detection ratio can be regarded as a reliable measure of the fault detection capability of a testing technique.

In our experiment, to compare metamorphic testing and assertion checking, we use the same test pool and only use the method level of mutation operators to produce mutants in procedural program style. Moreover, the target programs are deterministic; and thus, they produce the same output every time that a program executes a particular test case. Therefore, the failures shown in the outputs are also deterministic. However, our target programs are in Java, which is not the same as the C language. The set of mutation operators is not identical to that used by Andrews et al. [2]. On the other hand, many testing experiments use mutation analysis as the means to assure the effectiveness of various testing techniques.

To measure the time cost for applying MT and assertion checking, we use the time spent by individual subjects on individual target programs. We do not control how a subject conducts their tasks. Thus, a subject may make a mistake when doing a task, find out a similar mistake when working on another task, and then go back to the former task to rectify the first mistake. Thus, a preceding task may be over-estimated in terms of the time spent, while the later task may benefit from the development experience of the preceding task and be under-estimated. We treat this factor as random noise in the experiment. We measure the times reported by each subject on applying MT and on applying assertion checking.



7.4.4 Summary

The main contribution of this section is three-fold. (i) It is the first controlled experiment to compare metamorphic testing and assertion checking. The experiment shows that metamorphic testing is more effective than assertion checking as a means to identify faults. (ii) It provides the first empirical evidence to resolve the speculation whether subjects have difficulty formulating metamorphic relations and implementing MT. Indeed, the results of the experiment show that all subjects manage to propose metamorphic relations for the target programs after a brief general introduction on MT, and identical or very similar metamorphic relations are proposed by different subjects. (iii) This chapter further reports the first experiment to evaluate the effectiveness of (correct and faulty) metamorphic relations in MT. The result shows that a test suite can effectively identify failures from faulty target programs despite the presence of faulty metamorphic relation implementations. Our analysis on raw data also indicates that the granularity of using MT is coarser than assertion checking in failure detection.



Chapter 8

Conclusion

Software covers every corner of our lives, such as entertainment, education, and research. On the other hand, software faults are common. Everyone wants perfect software while most software is far from bug-free. Software debugging is important in software development. A major and time-consuming task in debugging is to locate faults. A common approach in statistical fault localization aims at locating program elements that are close to the faults, in terms of the positions (i.e., line numbers) of program statements. This relaxes the requirement to pinpoint the exact locations of the faults and has been shown empirically to be quite effective. In this thesis, we focus on such statistical fault localization techniques and investigate four topics.

First, we note that existing statistical fault-localization approaches focus on finding program elements that, when exercised, correlate strongly to program failures. However, an infected program state triggered by a fault may propagate a long way before the program execution finally causes a failure. Previous techniques are not effective to locate faults that, when exercised, have relatively weak correlations with program failures. We assess the suspiciousness scores of edges. We further set up a set of linear algebraic equations over the suspiciousness scores of basic blocks and statements, which abstractly model the propagation of suspicious program states through control flow edges in a back-tracing manner. Such an equation set can be efficiently solved using standard mathematical techniques such as Gaussian elimination and the least square method. Empirical results show that our proposed technique, known as CP, is effective when compared with existing techniques.

Second, during the experimental evaluation in a previous study, we find that representative predicate-level techniques are not as effective as statement-level techniques. We observe that the fault-localization capabilities of various evaluation sequences of the same Boolean expression are not identical.



Because of short-circuit evaluations of Boolean expressions in program execution, different evaluation sequences of a predicate may produce different resultant values. This inspires us to investigate the effectiveness of using Boolean expressions at the evaluation sequence level for statistical fault localization. Experiments on the Siemens programs and UNIX utility programs show that finer-grained fault-localization techniques may result in better effectiveness.

Third, in real-life testing and debugging environments, passed (i.e., successful) executions may not always be available or reliable. We propose a new model and develop a technique known as Slope, which uses two formulas that share the same model and can be applied to different scenarios with or without passed executions. Our underlying model first collects the execution counts of statements and, for each statement, calculates the fraction of failed executions with respect to all executions having the same execution count. It then calculates the failing rate accordingly. Considering every tuple of $\langle \text{failing rate, execution count} \rangle$ as a point in two-dimensional space, the model lines up these points and uses the slope of the line as the mean of the signal of suspiciousness and the fitting error as the noise to the signal. We have developed a formula based on the idea of signal-to-noise ratio and propose a novel and effective fault-localization technique. We continue to eliminate the dependency on passed executions by further approximations, which enables our Slope technique to work effectively in scenarios where passed executions are unavailable or unreliable to use. An experiment on UNIX utility programs shows that our approaches are promising.

Fourth, many previous studies overlook the statistical distributions of the spectra, on which their parametric techniques fully rely. We have argued and empirically verified that assuming a specific distribution of feature spectra of dynamic program statistics is problematic. It highlights a threat to construct validity in fault-localization research that previous studies do not report in their empirical evaluations and model development. We have proposed a non-parametric approach that applies general hypothesis testing techniques proposed by mathematicians to statistical fault localization, and cast our technique in a predicate-based setting. We have conducted experiments on the Siemens suite to evaluate the effectiveness of our model. The experimental results show that our model can be effective in locating faults and requires no artificial parameters or operators. Empirically, our approach gives better fault-localization effectiveness than existing predicate-level fault-localization techniques. In addition, we have also conducted experiment to compare the



effectiveness of using two standard non-parametric hypothesis testing methods and two standard parametric hypothesis testing methods. The results show that, when there is no evidence of any specific distribution of program feature spectra, the use of standard non-parametric hypothesis testing methods gives better results than the use of parametric hypothesis testing methods.

Since statistical fault localization uses dynamic execution information of programs and dependent partly on the pass/fail status of test cases, this thesis also discusses some related issues and reports a controlled experiment to study the application of metamorphic testing (MT) and assertion checking as the means to alleviate the test oracle problem. The experimental results indicate that, after training, testers can apply MT to test programs effectively, and that MT is a more effective testing strategy than assertion checking in terms of fault detection capability. Our study also reveals that the granularity of MRs is coarser than that of assertion checking, which may indicate that MRs provide a high level of abstraction for testers to deal with testing tasks.

In conclusion, this thesis develops statistical fault-localization techniques that are particularly effective when (a) the exercising of faulty statements are not strongly corrected to failures, (b) a program has many compound predicates, (c) passed executions are unavailable, or (d) the distribution profile of the program spectra is not available. The experiments in this thesis show that the proposed techniques are effective.

Future work may include integrating Slope with CP to gain better effectiveness and independence of passed executions. Second, since we use a concept of signal-to-noise ratio to construct the key formula, how to make use of an effective method to reduce the noise level is also a future direction. Third, we are also interested in locating multiple faults and locating faults in distributed systems. Fourth, we would like to conduct an empirical study to investigate the effect of incorrectly marked successful or failure-causing test cases on the effectiveness of fault localization techniques.



Bibliography

- [1] R. Abreu, P. Zoetewij, and A. J. C. van Gemund. “On the accuracy of spectrum-based fault localization”. In *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION (TAICPART-MUTATION 2007)*, pages 89-98. IEEE Computer Society Press, Los Alamitos, CA, 2007.
- [2] J. H. Andrews, L. C. Briand, and Y. Labiche. “Is mutation an appropriate tool for testing experiments?”. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 402-411. ACM Press, New York, NY, 2005.
- [3] S. Ar, M. Blum, B. Codenotti, and P. Gemmell. “Checking approximate computations over the reals”. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC 1993)*, pages 786-795. ACM Press, New York, NY, 1993.
- [4] P. Arumuga Nainar, T. Chen, J. Rosin, B. Liblit. “Statistical debugging using compound Boolean predicates”. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 5-15. ACM Press, New York, NY, 2007.
- [5] G. K. Baah, A. Podgurski, and M. J. Harrold. “The probabilistic program dependence graph and its application to fault diagnosis”. In *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 189-200. ACM Press, New York, NY, 2008.
- [6] T. Ball and S. Horwitz. “Slicing programs with arbitrary control-flow”. In *Proceedings of Workshop on Automated and Algorithm Debugging (AADEBUG 1993)*, volume 749 of *Lecture Notes in Computer Science*, pages 206-222. Springer, London, UK, 1993.
- [7] T. Ball, P. Mataga, and M. Sagiv. “Edge profiling versus path profiling: the showdown”. In *Proceedings of Symposium on Principles of Programming Languages (POPL 1998)*, pages 134-148. ACM Press, New York, NY, 1998.
- [8] Baudry, B., Fleurey, F., Le Traon, Y., 2006. “Improving test suites for efficient fault localization”. In *Proceedings of the 28th International*



Bibliography

- Conference on Software Engineering (ICSE 2006)*, pages 82-91. ACM Press, New York, NY.
- [9] B. Beizer. “Software testing techniques”. Van Nostrand Reinhold, New York, NY, 1990.
- [10] S. Beydeda. “Self-metamorphic-testing components”. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, volume 1, pages 265-272, IEEE Computer Society Press, Los Alamitos, CA, 2006.
- [11] R. V. Binder. “Testing object-oriented systems: models, patterns, and tools”. AddisonWesley, Reading, MA, 2000.
- [12] M. Blum and S. Kannan. “Designing programs that check their work”. *Journal of the ACM*, 42 (1): 269-291, 1995.
- [13] M. Blum, M. Luby, and R. Rubinfeld. “Self-testing/correcting with applications to numerical problems”. *Journal of Computer and System Sciences*, 47 (3): 549-595, 1993.
- [14] D. Binkley and M. Harman. “An empirical study of predicate dependence levels and trends”. In *Proceedings of International Conference on Software Engineering (ICSE 2003)*, pages 330-339. IEEE Computer Society, 2003.
- [15] M. D. Bond and K. S. McKinley. “Continuous path and edge profiling”. In *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2005)*, pages 130-140. IEEE Computer Society Press, Los Alamitos, CA, 2005.
- [16] J. F. Bowring, J.M. Rehg, and M. J. Harrold. “Active learning for automatic classification of software behavior”. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 195-205. ACM Press, New York, NY, 2004.
- [17] L. C. Briand, M. Di Penta, and Y. Labiche. “Assessing and improving state-based class testing: a series of experiments”. *IEEE Transactions on Software Engineering*, 30 (11): 770-783, 2004.
- [18] L. C. Briand, Y. Labiche, and X. Liu. “Using machine learning to support debugging with Tarantula”. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE 2007)*, pages 137-146. IEEE Computer Society, 2003.
- [19] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. “Application of metamorphic testing in numerical analysis”. In *Proceedings of the IASTED International Conference on Software Engineering (SE 1998)*, pages 191-197. ACTA Press, Calgary, Canada, 1998.
- [20] W. K. Chan, T.Y. Chen, S.C. Cheung, T. H. Tse, and Zhenyu Zhang. “Towards the testing of power-aware software applications for wireless



- sensor networks”. In Proceedings of the *12th International Conference on Reliable Software Technologies (Ada-Europe 2007)*, pages 84-99. LNCS 4498, Springer-Verlag, Berlin, 2007.
- [21] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. “A metamorphic approach to integration testing of context-sensitive middleware-based applications”. In Proceedings of *the 5th International Conference on Quality Software (QSIC 2005)*, pages 241-249. IEEE Computer Society Press, Los Alamitos, CA, 2005.
- [22] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. “Integration testing of context-sensitive middleware-based applications: a metamorphic approach”. *International Journal of Software Engineering and Knowledge Engineering*, 16 (5): 677-703, 2006.
- [23] W. K. Chan, M. Y. Cheng, S. C. Cheung, and T. H. Tse. “Automatic goal-oriented classification of failure behaviors for testing XML-based multimedia software applications: an experimental case study”. *Journal of Systems and Software*, 79 (5): 602-612, 2006.
- [24] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. “Towards a metamorphic testing methodology for service-oriented software applications”. *The 1st International Conference on Services Engineering (SEIW 2005)*. In Proceedings of *the 5th International Conference on Quality Software (QSIC 2005)*, pages 470-476. IEEE Computer Society Press, Los Alamitos, CA, 2005.
- [25] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. “A metamorphic testing approach for online testing of service-oriented software applications”. *International Journal of Web Services Research*, 4 (2): 60-80, 2007.
- [26] W. K. Chan, S. C. Cheung, J. C. F. Ho, and T. H. Tse. “PAT: a pattern classification approach to automatic reference oracles for the testing of mesh simplification programs”. *Journal of Systems and Software* 82(3): 422-423, 2008.
- [27] W. K. Chan, J. C. F. Ho, and T. H. Tse. “Finding failures from passed test cases: improving the pattern classification approach to the testing of mesh simplification programs”, *Software Testing, Verification and Reliability* (2009). doi:10.1002/stvr.408.
- [28] W. K. Chan, J. C. F. Ho, and T. H. Tse. “Piping classification to metamorphic testing: an empirical study towards better effectiveness for the identification of failures in mesh simplification programs”. In Proceedings of *the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 397-404. IEEE Computer Society Press, Los Alamitos, CA, 2007.
- [29] W. K. Chan, L. Mei, and Z. Zhang. “Modeling and testing of cloud applications”, in Proceedings of 2009 IEEE Asia-Pacific Services



Bibliography

- Computing Conference (APSCC 2009), IEEE Computer Society Press, Los Alamitos, CA, pp. 111-118 (2009).
- [30] D. Chapman. “A program testing assistant”. *Communications of the ACM*, 25 (9): 625-634, 1982.
- [31] T. Y. Chen, J. Feng, and T. H. Tse. “Metamorphic testing of programs on partial differential equations: a case study”. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2002)*, pages 327-333. IEEE Computer Society Press, Los Alamitos, CA, 2002.
- [32] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. “Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing”. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 191-195. ACM Press, New York, NY, 2002.
- [33] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. “Fault-based testing without the need of oracles”. *Information and Software Technology*, 45 (1): 1-9, 2003.
- [34] W. Chen, R. H. Untch, G. Rothermel, S. Elbaum, and J. von Ronne, “Can fault-exposure-potential estimates improve the fault detection abilities of test suites?”. *Software Testing, Verification and Reliability* 12 (4): 197-218 (2002).
- [35] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, K. Vaswani. “Holmes: effective statistical debugging via efficient path profiling”. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 34-44. ACM Press, New York, NY, 2009.
- [36] H. Cleve, A. Zeller. “Locating causes of program failures”. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 342-351. ACM Press, New York, NY.
- [37] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke. “User guidance for creating precise and accessible property specifications”. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2006/FSE-14)*, pages 208-218. ACM Press, New York, NY, 2006.
- [38] J. S. Collofello and L. Cousins. “Towards automatic software fault location through decision-to-decision path analysis”. In *Proceedings of the 1987 National Computer Conference*, pages 539–544. Chicago, IL, 1987.
- [39] G. E. Dallal. “Why $P = 0.05$?”. Available at <http://www.tufts.edu/gdallal/p05.htm>.



- [40] H. Do, S. G. Elbaum, G. Rothermel. “Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact”. *Empirical Software Engineering* 10 (4), 405–435.
- [41] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, “An empirical study of the effect of time constraints on the cost-benefits of regression testing”. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2008/FSE-16)*, pages 71-82. ACM Press, New York, NY, 2008.
- [42] H. Do and G. Rothermel, “A controlled experiment assessing test case prioritization techniques via mutation faults”. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 411-420. IEEE Computer Society Press, Los Alamitos, 2005.
- [43] H. Do and G. Rothermel, “An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models”. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2006/FSE-14)*, pages 141-151. ACM Press, New York, NY, 2006.
- [44] H. Do and G. Rothermel, “On the use of mutation faults in empirical assessments of test case prioritization techniques”, *IEEE Transactions on Software Engineering* 32 (9): 733-752 (2006).
- [45] H. Do and G. Rothermel, “Using sensitivity analysis to create simplified economic models for regression testing”, in *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 51-62. ACM Press, New York, NY, 2008.
- [46] H. Do, G. Rothermel, and A. Kinneer, “Prioritizing JUnit test cases: an empirical assessment and cost-benefits analysis”, *Empirical Software Engineering* 11: 33-70 (2006).
- [47] J. A. Durães and H. S. Madeira. “Emulation of software faults: A field data study and a practical approach”. *IEEE Transactions on Software Engineering*, 32 (11): 849–867, 2006.
- [48] S. Elbaum, D. Gable, and G. Rothermel, “The impact of software evolution on code coverage information”, in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, page 170. IEEE Computer Society Press, Los Alamitos, CA2001.
- [49] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, “Prioritizing test cases for regression testing”, in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, ACM SIGSOFT Software Engineering Notes 25 (5): 102-112 (2000).



Bibliography

- [50] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, “Incorporating varying test costs and fault severities into test case prioritization”. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 329-338. IEEE Computer Society Press, Los Alamitos, CA, 2001.
- [51] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, “Test case prioritization: a family of empirical studies”, *IEEE Transactions on Software Engineering* 28 (2): 159-182 (2002).
- [52] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. “Selecting a cost-effective test case prioritization technique”. *Software Quality Control*, 12 (3): 185-210, 2004.
- [53] P. Francis, D. Leon, M. Minch, and A. Podgurski. “Tree-based methods for classifying software failures”. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 451- 462. IEEE Computer Society Press, Los Alamitos, CA, 2004.
- [54] A. Griesmayer, S. Staber, R. Bloem. “Automated fault localization for C programs”. *Electronic Notes in Theoretical Computer Science* 174 (4), 95–111, 2007.
- [55] A. Gotlieb and B. Botella. “Automated metamorphic testing”. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, pages 34-40. IEEE Computer Society Press, Los Alamitos, CA, 2003.
- [56] N. Gupta, H. He, X. Zhang, and R. Gupta. “Locating faulty code using failure-inducing chops”. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 263–272. ACM Press, New York, NY, 2005.
- [57] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. “An empirical investigation of the relationship between spectra differences and regression faults”. *Software Testing, Verification and Reliability*, 10 (3): 171–194, 2000.
- [58] R. Helm, W. M. Holland, and D. Gangopadhyay. “Contracts: specifying behavioral compositions in object-oriented systems”. In *Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1990)*, ACM SIGPLAN Notices, 25 (10): 169-180, 1990.
- [59] W. E. Howden. “Weak mutation testing and completeness of test sets”. *IEEE Transactions on Software Engineering*, SE-8 (4): 371-379, 1982.
- [60] P. Hu, “Automated Fault Localization: a Statistical Predicate Analysis Approach”. PhD Thesis, The University of Hong Kong, 2006.
- [61] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse. “An empirical comparison between direct and indirect test result checking



- approaches”. In Proceedings of *the 3rd International Workshop on Software Quality Assurance (SOQUA 2006)* in conjunction with *the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT 2006/FSE-14)*, pages 6-13. ACM Press, New York, NY, 2006.
- [62] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse, “Fault localization with non-parametric program behavior model”, in Proceedings of *the 8th International Conference on Quality Software (QSIC 2008)*, pages 385-395. IEEE Computer Society Press, Los Alamitos, California, 2008.
- [63] M. Hutchins, H. Foster, T. Goradia, T. Ostrand. “Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria”. In Proceedings of *the 16th International Conference on Software Engineering (ICSE 1994)*, pages 191-200. IEEE Computer Society Press, Los Alamitos, CA.
- [64] D. Jeffrey, N. Gupta, R. Gupta. “Fault localization using value replacement”. In Proceedings of *the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 167-178. ACM Press, New York, NY, 2008.
- [65] B. Jiang, W. K. Chan, Z. Zhang, and T. H. Tse. “Where to Adapt Dynamic Service Compositions”, in *Proceedings of the 18th International World Wide Web Conference (WWW 2009)*, Madrid, Spain, Apr 20 - 24, 2009. (poster track)
- [66] B. Jiang, Z. Zhang, W. K. Chan and T. H. Tse, “Adaptive random test case prioritization”, in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 233-244.
- [67] B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen, “How well do test case prioritization techniques support statistical fault localization”. In Proceedings of *the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*, vol. 1, pages 99-106. IEEE Computer Society Press, Los Alamitos, CA, 2009.
- [68] J. A. Jones and M. J. Harrold. “Empirical evaluation of the Tarantula automatic fault-localization technique”. In Proceedings of *the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 273–282. ACM Press, New York, NY, 2005.
- [69] J. A. Jones, M. J. Harrold, J. F. Bowring. “Debugging in parallel”. In Proceedings of *the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 16-26. ACM Press, New York, NY, 2007.



Bibliography

- [70] J. A. Jones, M. J. Harrold, J. Stasko. “Visualization of test information to assist fault localization”. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 467-477. ACM Press, New York, NY, 2002.
- [71] A. J. Ko and B. A. Myers. “Debugging reinvented: asking and answering why and why not questions about program behavior”. In *Proceedings of International Conference on Software Engineering (ICSE 2008)*, pages 301-310. ACM Press, New York, NY, 2008.
- [72] B. Korel. “PELAS: Program error-locating assistant system”. *IEEE Transactions on Software Engineering*, 14 (9): 1253–1260, 1988.
- [73] M. Last, M. Friedman, and A. Kandel. “The data mining approach to automated software testing”. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003)*, pages 388-396. ACM Press, New York, NY, 2003.
- [74] B. Liblit, A. Aiken, A.X. Zheng, and M. We. Jordan. “Bug isolation via remote program sampling”. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*, ACM SIGPLAN Notices, 38 (5): 141–154, 2003.
- [75] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. Jordan. “Scalable statistical bug isolation”. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, ACM SIGPLAN Notices 40 (6), 15–26.
- [76] C. Liu, L. Fei, X. Yan, S. P. Midkiff, J. Han. “Statistical debugging: a hypothesis testing-based approach”. *IEEE Transactions on Software Engineering* 32 (10), 831–848.
- [77] C. Liu, X. Yan, L. Fei, J. Han, S. P. Midkiff. “SOBER: statistical model-based bug localization”. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundation of Software Engineering (ESEC 2005/FSE-13)*, ACM SIGSOFT Software Engineering Notes 30 (5), 286–295.
- [78] R. Lowry. “Concepts and Applications of Inferential Statistics”. Vassar College, Poughkeepsie, NY, 2006. Available at <http://faculty.vassar.edu/lowry/webtext.html>.
- [79] H. Lu, W. K. Chan, and T. H. Tse. “Testing context-aware middleware centric programs: a data flow approach and an RFID-based experimentation”. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2006/FSE-14)*, pages 242-252. ACM Press, New York, NY, 2006.



- [80] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon. “MuJava: an automated class mutation system”. *Software Testing, Verification and Reliability*, 15 (2): 97-133, 2005.
- [81] L. I. Manolache and D. G. Kourie. “Software testing using model programs”. *Software: Practice and Experience*, 31 (13): 1211-1236, 2001.
- [82] L. Mei, Z. Zhang, and W. K. Chan, “More Tales of Clouds: Software Engineering Research Issues from the Cloud Application Perspective”, in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*, vol. 1, IEEE Computer Society Press, Los Alamitos, CA, pp. 525-530 (2009). (short paper)
- [83] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse, “Test case prioritization for regression testing of service-oriented business applications”. In *Proceedings of the 18th International World Wide Web Conference (WWW 2009)*, pages 901-910. ACM Press, New York, NY, 2009.
- [84] B. Meyer. “Applying ‘design by contract’”. *IEEE Computer*, 25 (10): 40-51, 1992.
- [85] C. Murphy. “Using runtime testing to detect defects in applications without test oracles”. In *Companion to Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2008/FSE-16)*, pages 21-24. ACM Press, New York, NY, 2008.
- [86] A. S. Namin, J. H. Andrews, and D. J. Murdoch. “Sufficient mutation operators for measuring test effectiveness”. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 351-360, ACM Press, New York, NY, 2008.
- [87] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. “An experimental determination of sufficient mutant operators”. *ACM Transactions on Software Engineering and Methodology*, 5 (2): 99-118, 1996.
- [88] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. “Automated support for classifying software failure reports”. In *Proceedings of International Conference on Software Engineering (ICSE 2003)*, pages 465-475. IEEE Computer Society, 2003.
- [89] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. “A controlled experiment in maintenance comparing design patterns to simpler solutions”. *IEEE Transactions on Software Engineering*, 27 (12): 1134- 1144, 2001.
- [90] M. Renieris, S. P. Reiss. “Fault localization with nearest neighbor queries”. In *Proceedings of the 18th IEEE International Conference on*



Bibliography

- Automated Software Engineering (ASE 2003)*, pages 30-39. IEEE Computer Society Press, Los Alamitos, CA.
- [91] T. Reps, T. Ball, M. Das, and J. Larus. “The use of program profiling for software maintenance with applications to the year 2000 problem”. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC 1997)*. Volume 1301 of LNCS, Springer-Verlag, 1997, pp. 432-449.
- [92] G. Rothermel, S.G. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. “The impact of test suite granularity on the cost- effectiveness of regression testing”. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 130-140. ACM Press, New York, NY, 2002.
- [93] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. “Test case prioritization: an empirical study”, in *Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM 1999)*, pages 179-188. IEEE Computer Society Press, Los Alamitos, CA, 1999.
- [94] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. “Prioritizing test cases for regression testing”. *IEEE Transactions on Software Engineering* 27 (10): 929-948 (2001).
- [95] Eduard Säckinger, “Broadband circuits for optical fiber communication”. Wiley-Interscience, 2005.
- [96] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. “Lightweight fault localization using multiple coverage types”. In *Proceedings of International Conference on Software Engineering (ICSE 2009)*, pages 56-66. IEEE Computer Society Press, Los Alamitos, CA, 2009.
- [97] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. Harrold. “Fault localization and repair for Java runtime exceptions”. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 153-164. ACM Press, New York, NY, 2009.
- [98] Y. Sun and E. L. Jones. “Specification-driven automated testing of GUI-based Java programs”. In *Proceedings of the 42nd Annual Southeast Regional Conference (ACM-SE 42)*, pages 140-145. ACM Press, New York, NY, 2004.
- [99] R. N. Taylor. “Assertions in programming languages”. *ACM SIGPLAN Notices*, 15 (1): 105-114, 1980.
- [100] F. Tip. “A survey of program slicing techniques”. *Journal of Programming Languages*, 3 (3): 121–189, 1995.
- [101] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. “Testing context sensitive middleware-based software applications”. In *Proceedings of the 28th Annual International Computer Software and*



- Applications Conference (COMPSAC 2004)*, volume 1, pages 458-465. IEEE Computer Society Press, Los Alamitos, CA, 2004.
- [102] M. Vanmali, M. Last, and A. Kandel. “Using a neural network in the software testing process”. *International Journal of Intelligent Systems*, 17 (1): 45-62, 2002.
- [103] I. Vessey. “Expertise in debugging computer programs: a process analysis”. *International Journal of Man-Machine Studies* 23 (5), 459–494.
- [104] J. M. Voas. “PIE: a dynamic failure-based technique”. *IEEE Transactions on Software Engineering*, 18 (8): 717-727, 1992.
- [105] M. Vokáč, W. Tichy, D. We. K. Sjoberg, E. Arisholm, and M. Aldrin. “A controlled experiment comparing the maintainability of program designed with and without design patterns: a replication in a real programming environment”. *Empirical Software Engineering*, 9 (3):149-195, 2004.
- [106] X. Wang, S. C. Cheung, W. K. Chan, Z. Zhang. “Taming coincidental correctness: refine code coverage with context pattern to improve fault localization”. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 45-55. ACM Press, New York, NY, 2009.
- [107] M. Weiser. “Program slicing”. *IEEE Transactions on Software Engineering*, SE-10 (4): 352–357, 1984.
- [108] E. J. Weyuker. “On testing non-testable programs”. *The Computer Journal*, 25 (4): 465-470, 1982.
- [109] E. Wong, Y. Qi, L. Zhao, and K. Cai. “Effective Fault Localization using Code Coverage”. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 449–456, IEEE Computer Society, Washington, DC, USA, 2007.
- [110] P. Wu. “Iterative metamorphic testing”. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, volume 1, pages 19-24. IEEE Computer Society Press, Los Alamitos, CA, 2005
- [111] Q. Xie and A. M. Memon. “Designing and comparing automated test oracles for GUI-based software applications”. *ACM Transactions on Software Engineering and Methodology*, 16 (1): Article No. 4, 2007.
- [112] Y. Yu, J. A. Jones, M. J. Harrold. “An empirical study of the effects of test-suite reduction on fault localization”. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 201-210. ACM Press, New York, NY, 2009.
- [113] A. Zeller. “Isolating cause-effect chains from computer programs”. In *Proceedings of the 10th ACM SIGSOFT International Symposium on*



Bibliography

- Foundations of Software Engineering (SIGSOFT 2002/FSE-10)*. ACM SIGSOFT Software Engineering Notes, 27 (6): 1–10, 2002.
- [114] A. Zeller, R. Hildebrandt. “Simplifying and isolating failure-inducing input”. *IEEE Transactions on Software Engineering* 28 (2), 183–200.
- [115] X. Zhang, N. Gupta, and R. Gupta. “Locating faults through automated predicate switching”. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 272–281. ACM Press, New York, NY, 2006.
- [116] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. “Towards locating execution omission errors”. In *Proceedings of Programming Language Design and Implementation (PLDI 2007)*, pages 415–424. ACM Press, 2007.
- [117] Z. Zhang, W. K. Chan, and T. H. Tse. “Synthesizing component-based WSN applications via automatic combination of code optimization techniques”. In *Proceedings of the 7th International Conference on Quality Software (QSIC 2007)*, pages 181–190. IEEE Computer Society Press, Los Alamitos, CA, USA, 2007.
- [118] Z. Zhang, W. K. Chan, T. H. Tse, and P. Hu. “Experimental study to compare the use of metamorphic testing and assertion checking”, *Journal of Software (JoS)* 20(10), 2009.
- [119] Z. Zhang, W. K. Chan, T. H. Tse, P. Hu, and X. Wang. “Is non-parametric hypothesis testing model robust for statistical fault localization?”. In *Journal of Information and Software Technology (IST)* 51(11): 1573–1585 (2009).
- [120] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. “Capturing propagation of infected program states”. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC 7/FSE-17)*, pages 43–52. ACM Press, New York, NY, 2009.
- [121] Z. Zhang, W. K. Chan, T. H. Tse, H. Lu, and L. Mei. “Resource prioritization of code optimization techniques for program synthesis of wireless sensor network applications”. *Journal of Systems and Software (JSS)* 82(9): 1376–1387 (2009).
- [122] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse. “Debugging through evaluation sequences: a controlled experimental study”. In *Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008)*, pages 128–135. IEEE Computer Society Press, Los Alamitos, CA, 2008.
- [123] Z. Zhang, B. Jiang, W. K. Chan, T. H. Tse, and Xinming Wang. “Fault localization through evaluation sequences”. In *Journal of Systems and Software (JSS)* 83(2): 174–187 (2010).



- [124] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, A. Aiken. “Statistical debugging: simultaneous identification of multiple bugs”. In *Proceedings of the 23rd International Conference on Machine Learning (ICML 2006)*, pages 1105-1112. ACM Press, New York, NY, 2006.
- [125] D. G. Zill, M. R. Cullen. “Advanced Engineering Mathematics”. Jones and Bartlett Publishers, Sudbury, MA.



Bibliography
