석사 학위논문
Master's Thesis

프로그램의 동적 행동을 이용한 효과적인 소프트웨어 결함 위치추정

Effective Software Fault Localization using Dynamic Program Behaviors

문 석 현 (文 晳 鉉 Moon, Seokhyeon)
전산학과
Department of Computer Science

KAIST

2014

프로그램의 동적 행동을 이용한 효과적인 소프트웨어 결함 위치추정

Effective Software Fault Localization using Dynamic Program Behaviors

Effecitve Software Fault Localization using Dynamic Program Behaviors

Advisor : Professor Kim, Moonzoo

by

Moon, Seokhyeon

Department of Computer Science

KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science . The study was conducted in accordance with Code of Research Ethics[1].

2013. 12. 17.

Approved by

Professor Kim, Moonzoo

[Advisor]

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

프로그램의 동적 행동을 이용한 효과적인 소프트웨어 결함 위치추정

문 석 현

위 논문은 한국과학기술원 석사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2013년 12월 17일

심사위원장 김 문 주 (인)

심사위원 류 석 영 (인)

심사위원 양 현 승 (인)

MCS
20123239

문 석 현. Moon, Seokhyeon. Effective Software Fault Localization using Dynamic Program Behaviors. 프로그램의 동적 행동을 이용한 효과적인 소프트웨어 결함 위치 추정. Department of Computer Science . 2014. 49p. Advisor Prof. Kim, Moonzoo. Text in English.

ABSTRACT

As software becomes more complex due to a number of complex software requirements, debugging program becomes very challenging. One of the most expensive task of debugging process is to locate the root cause of program failures (i.e., fault). This process, called *fault localization*, requires developers to understand complex internal logic of Program Under Test (PUT) to identify the location of fault while inspecting enormous program states.

To reduce the human efforts for fault localization, therefore, we propose two fault localization techniques that utilize dynamic program behaviors. Each of the proposed techniques automatically ranks statements of PUT according to their predicted risk of containing faults, which is essentially computed based on the dynamic information (e.g., a set statements executed by a test case) gathered from test case executions of PUT. Developers can effectively locate the fault by inspecting PUT while following the order of statements in the ranking generated by the proposed techniques. The key ideas of proposed techniques can be summarized as follows.

One of the proposed techniques, called FIESTA, utilizes *correlation* between failing executions and each of executed program statements to identify the location of faults. The degree of the correlation indicates the degree of possibility that a statement contains faults, which is used to rank program statements. FIESTA strengthens the correlation between failing executions and faulty statements, by eliminating program structures that can weaken the correlation through the program transformation. It also assigns fault weight on a test case that indicates likelihood of the test case to execute a faulty statement. Thanks to the strengthened correlation, FIESTA can localize faulty statements more effectively than the previous fault localization techniques do.

The other one, called MUSE, utilizes *mutation analysis* to uniquely capture the relation between individual program statements and observed failures. The key idea of MUSE is to identify a faulty statement by utilizing different characteristics of two groups of mutants — one that mutates a faulty statement and the other that mutates a correct one. For example, MUSE prioritizes statements where modifications (i.e., mutants) on those statements make failing test cases become passing ones, since a faulty program is usually repaired by modifying (i.e., mutating) faulty statements.

In addition, we also propose an evaluation metric for fault localization techniques based on information theory, called Locality Information Loss (LIL). It enables to measure the aptitude of a fault localization technique for tools that automatically fix faults, which the existing evaluation metric cannot.

To evaluate the effectiveness of our techniques, we carried out experiments by applying the proposed techniques on 12 C-programs including 5 real-world programs. The experimental results showed that the proposed techniques are very effective for locating many faults. For example, MUSE ranks a faulty statement among the top 7.4 places on average, which is about 25 times more precise than the state-of-the-art fault localization technique, called Op2, on average. It implies that the proposed techniques can reduce the human efforts for fault localization significantly.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

As the software development is primarily done by human developers, software inevitably contains bugs which are produced by human mistakes. Software bugs not only make software deviates the required functionalities, but also cause severe impacts on our lives. According to the report of National Institute of Standards and Technology (NIST) in 2002, "software bugs, or errors, are so prevalent and so detrimental that they cost the US economy an estimated $59 billion annually, or about 0.6 percent of the gross domestic product" [48]. As a result, many human developers have examined the correctness of software products to reduce the number of bugs in their products.

However, debugging program becomes challenging as the complexity of software rapidly increases due to a number of complex software requirements [60]. One of the most expensive and laborious task of debugging activity is to locate the root cause of program failures [33, 66], which is called *fault localization*. This process requires human developers to understand the complex internal logic of the Program Under Test (PUT) and reason about the differences between passing and failing executions in order to identify the location of fault. Therefore, reducing the human efforts for fault localization will significantly decrease the total cost of whole debugging activity.

This dissertation presents novel techniques to assist developers effectively locate faulty statements in PUT. It presents two automated fault localization techniques (called FIESTA and MUSE), each of which automatically ranks program statements of PUT according to their predicted risk of containing faults. Each of the proposed techniques respectively utilizes test case executions of transformed PUT (Chapter 3) and those of mutated PUT (Chapter 4), to compute the predicted risk of each statement. Developers can effectively locate faults by inspecting PUT while following the order of statements in the ranking generated by the proposed techniques. In addition, the dissertation also presents a new evaluation metric for fault localization techniques, that enables to measure the aptitude of fault localization techniques for tools that automatically fix faults [68], which is not possible by the exiting evaluation metric [56] (Chapter 5).

From the evaluation results of the proposed techniques on 12 C-programs including 5 real-world programs, we confirmed that the proposed techniques not only outperform the previous fault localization techniques, but also can be a practical solution for fault localization. For example, in order to locate a faulty statement, the proposed technique MUSE requires a developer to inspect 7.4 statements on average, whereas the state-of-art spectrum-based fault localization technique Op2 does 184.6 statements, for our subject programs.

In this chapter, we describe the limitations of the previous automated fault localization techniques, and the key ideas of our approaches that can mitigate the described limitations. We then present the summary of the contributions of this work.

1.1 Limitations of Previous Approaches

A number of automated fault localization techniques have been proposed to reduce the developers' manual efforts for fault localization [6, 19, 31, 36, 56, 70, 78]. Among them, a promising research direction is Spectrum-Based Fault Localization (SBFL) [36, 38, 42, 47, 72]. SBFL uses program spectrum, i.e.

summarized profile of test suite executions that indicate which parts of PUT are executed during program executions, to rank program entities (such as statements or branches) according to their predicted risk of containing faults, called suspiciousness. The human developer, then, is to inspect PUT following the order of entities in the given ranking, in the hope that the faulty entity will be encountered near the top of the ranking [71].

The key idea of SBFL is to utilize the correlation between failing executions and executed faulty statements to identify the locations of faulty statements. SBFL collects program spectrum from both failing executions (i.e., executions that do not produce expected output) and passing executions (i.e., executions that produce expected output), and then gives high suspiciousness to statements that are strongly correlated with failing executions. In other words, SBFL expects the correlation between faulty statements and failing executions is stronger than the correlation between correct statements and failing executions.

Although SBFL has empirically been proven to outperform other kinds of fault localization techniques [35], it has also been criticized for their impractical accuracy [54]. One of main causes for the impractical accuracy is the existence of *Coincidentally Correct Test cases (CCTs)*, which are passing test cases despite executing the faulty statements [58]. Due to the existence of CCTs, the correlation between faulty statements and failing executions can be not strong enough to identify the locations of faulty statements effectively [15, 67, 73]. This is because faulty statements executed by CCTs are often considered as non-faulty ones since they are executed by passing test cases. Thus, the effectiveness of SBFL techniques can decrease due to the CCTs.

In addition to the negative effect of CCTs, the coarse granularity of block level of SBFL is another reason for the impractical accuracy [33]. The program spectrum used by SBFL techniques is simply a combination of the control flow of PUT and the results from test cases. Consequently, all statements in the same basic block share the same spectrum and, therefore, the same ranking. This often inflates the number of statements needed to be inspected before encountering the fault.

In the view point of the technique evaluation, on the other hand, the traditional evaluation metric in SBFL literature is the Expense metric, which is the percentage of program statements the human developer needs to inspect before encountering the faulty one [56]. However, recent work showed that the expense metric failed to account for the performance of the automated program repair tool that used various SBFL techniques to locate the fix: techniques proven to rank the faulty statement higher than others actually performed poorer when used in conjunction with a repair tool [55].

1.2 Our Approaches

We propose two automated fault localization techniques, which mitigate and overcome the described limitations of SBFL respectively. In addition, we also propose a new evaluation metric for fault localization techniques that overcomes the limitation of existing evaluation metric.

1.2.1 FIESTA: Fault Localization to Mitigate the Negative Effect of CCTs

We propose a new fault localization technique, called Fault-weIght and atomizEd condition baSed local-izaTion Algorithm (FIESTA), that mitigates the negative effect of CCTS in the following ways: (1) To transform a target program to reduce the number of CCTs (Debuggability transformation), and (2) To develop a new suspiciousness formula that mitigates the negative effect of CCTs approximately.

First, FIESTA transforms a target program P into the semantically equivalent program P' by converting a compound conditional statement into nested conditional statements with atomic conditions. Through this Debuggability transformation, the precision of SBFL technique can increase due to the decreased number of CCTs. Second, FIESTA defines a new suspiciousness metric that reduces the negative of CCTs by considering fault weights on test cases. We define a fault weight on a test case t to indicate *likelihood* of t to execute a faulty statement and utilize the fault weight values of the test cases that cover a target statement s to compute a suspiciousness of s.

1.2.2 MUSE: Mutating Faulty Programs for Fault Localization

We also present another new fault localization technique, called MUSE (MUtation-baSEd fault localization technique), that overcomes the coarse granularity of block level of SBFL. MUSE uses mutation analysis to uniquely capture the relation between individual program statements and the observed failures. It is free from the coercion of shared ranking from the block structure. The basic mutation testing is defined as artificial injection of syntactic faults [17]. However, we focus on what happens when we mutate an already faulty program and, particularly, the faulty program statement. Intuitively, since a faulty program can be repaired by modifying faulty statements, mutating (i.e., modifying) faulty statements will make more failed test cases pass than mutating correct statements. In contrast, mutating correct statements will make more passed test cases fail than mutating faulty statements. This is because mutating correct statements introduces new faulty statements in addition to the existing faulty statements in a PUT. These two observations form the basis of our new fault localization technique.

1.2.3 LIL: An Evaluation Metric for Fault Localization Techniques

To overcome the limitation of exiting evaluation metric for fault localization techniques, we propose a new evaluation metric that is not tied to the ranking model. Our new evaluation metric, Locality Information Loss (LIL), actually measures the loss of information between the true locality of the fault and the predicted locality from a localization technique, using information theory. It can measure the aptitude of a localization technique for automated fault repair systems as well as human debuggers. In addition, it can be applied to any fault localization techniques (not just SBFL) and to describe localization of any number of faults.

1.3 Contributions

The proposed research will provide the following contributions:

- Techniques (i.e., MUSE and FIESTA) that practically assist developers locate faulty statements effectively.
  - A fault localization technique, called Fault-weIght and atomizEd condition baSed local-izaTion Algorithm (FIESTA), improves the precision of fault localization based on two proposed techniques: (1) Debuggability transformation, that improves the precision by reducing the number of CCTs, and (2) fault weight based suspiciousness metric, that improves the precision by utilizing the likelihood of each test case to execute faulty statements.

  - A fault localization technique, called MUtation-baSEd fault localization (MUSE), that significantly improves the precision of fault localization by uniquely capturing the relation between individual program statements and the observed failures based on mutation analysis.

- An evaluation metric for fault localization techniques, called Locality Information Loss (LIL), that can measure the aptitude of a fault localization technique for automated program repair tools based on information theory.

- Empirical demonstration of effectiveness and practicality of proposed techniques on 12 C-programs including 5 real-world programs.

The contributions of this dissertation are also supported by the following papers:

- **S.Moon**, Y.Kim, M. Kim, S. Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization, The 7th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2014), to appear (**acceptance rate: 28%**).

- **S.Moon**, Y.Kim, M. Kim, FEAST: An Enhanced Fault Localization Technique using Probability of Test Cases Executing Faults, Journal of KIISE: Software and Applications, Vol 40, Num 10, Oct 2013 (**invited paper**).

- **S.Moon**, Y.Kim, M. Kim, An Enhanced Fault Localization Technique using Probability of Test Cases Executing Faults, Korea Conference on Software Engineering (KCSE), Jan 30 – Feb 1, 2013.

- **S.Moon**, Y.Kim, M. Kim, FIESTA: Effective Fault Localization to Mitigate the Negative Effect of Coincidentally Correct Tests, in preparation.

It should be noted that Locality Information Loss (LIL) metric is originally proposed by prof. Shin Yoo at University of College London, who has collaborated with us for developing new fault localization techniques.

1.4 Outline of the Dissertation

Chapter 2 describes background and related work. We first define certain terminologies. We then present related work and the limitations of previous approaches.

Chapter 3 presents our technique FIESTA, that mitigates the negative effect of CCTs. We first explain the intuitions of FIESTA, and then present detailed approaches. We next show the high effectiveness of FIESTA through the empirical evaluation. The empirical evaluation compares the effectiveness of FIESTA with the state-of-art SBFL techniques. Finally, we discuss the limitations of FIESTA, and related work of FIESTA.

Chapter 4 presents our technique MUSE, that overcomes the coarse granularity of block level of SBFL. We first deliver intuitions of MUSE, and then present detailed approaches. We next demonstrate the superior precision of MUSE through the empirical evaluation. The empirical evaluation compares the effectiveness of MUSE with the state-of-art SBFL techniques. Finally, we discuss the reasons of the superior precision of MUSE, and related work of MUSE.

Chapter 5 presents LIL metric, an metric for fault localization techniques based on information theory, which can measure the aptitude of fault localization techniques toward automated program repair tools. We evaluate fault localization techniques using LIL, and then discuss the advantages of LIL

observed through the evaluation. In addition, a case study with an automated program repair tool [27] is presented to demonstrate that LIL metric can be used to predict which fault localization techniques help automated program repair tools to improve their performance.

Chapter 6 finally concludes this dissertation with future work.

Chapter 2. Background and Related Work

This chapter provides background and related work. We describe previous debugging techniques including automated fault localization techniques, and their limitations.

2.1 Definitions

In this dissertation, certain terminologies will be used repeatedly. We follow certain definitions in the document, IEEE Standard 729 Glossary for Software Engineering Technology [63], which provides definitions of fault, error, failure, and debug as follows:

- *Fault*: an incorrect step, process, or data definition in a computer program.
- *Error*: the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.
- *Failure*: the inability of a system or component to perform its required functions within specified performance requirements
- *Debug*: to detect, locate, and correct faults in a computer program.

Thus, a developer makes faults, which can can cause errors. Errors can make one detects failures. A developer debug (i.e., debugging) Program Under Test (PUT) by following the process: (1) detecting faults in PUT via failures, (2) *locating faults (i.e., fault localization)*, and (3) correcting the faults.

We also define a few terminologies, which help readers understand our following work:

- *Test case*: A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
- *Passing test case*: A test case that produces an expected output.
- *Failing test case*: A test case that does not produce an expected output.

2.2 Traditional Debugging

In traditional debugging approach, developers usually insert 'print' statements into several places of PUT, which are likely to indicate locations of faults, based on their intuitions. Then, developers attempt to identify locations of faults by analyzing outputs of each inserted print statement in failing executions. Although this approach has been used widely, it can consume a lot of human efforts because the approach requires in-depth understanding of PUT to insert print statements in appropriate places and to analyze the outputs of the inserted print statements.

Another way to find the locations of faults is to use symbolic debuggers such as GDB [2] and DBX [40]. These debuggers assist developers to trace internal states of program executions by supporting

break pointing, stepping, and state modifying features. Developers can examine internal states of failing executions on several places of PUT by using the supported features, without manually inserting print statements. These examinations may enable developers to detect abnormal internal states, which can indicate the locations of faults. However, debugging becomes challenging as the complexity of program increases [60], even if one uses those symbolic debuggers. Developers should manually insert break points to PUT, and examine enormous internal program states of failing executions to locate faults.

2.3 Program Slicing

Another kind of debugging techniques uses programs slices. Program slicing techniques [65] identify particular parts of PUT that could affect a value of a variable at a specific program statement. The particular parts are called a *slice*, and the variable with the program statement is called a *slicing criterion*. Slicing approaches compute a slice, which could affect the slicing criterion. In the view point of program debugging, the slice will be the particular parts of PUT that contain faults if the slicing criterion is specified as the one that manifests failures, and the slice is examined by a developer to locate faults. Slicing techniques are classified with static-slicing, execution-slicing, and dynamic-slicing according to the information they use for the computation of a slice.

2.3.1 Static Slicing

Weiser proposed a slicing approach that computes a slice by analyzing PUT statically [69, 70]. Given a slicing criterion in PUT, a computed slice is a set of all transitively relevant statements to the slicing criterion, according to static control dependencies and data dependencies of PUT [65]. As the Weiser's approach utilizes only the static information to compute a slice, this approach is called static slicing.

Since static slicing can reduce the number of statements in PUT that need to be examined to localize faults, it could reduce the human efforts for fault localization. However, as the static slicing approach does not utilize any dynamic information from the executions that actually cause failures, the computed slice is likely to include a number of statements that actually do not affect the slicing criterion (i.e., abnormal program behaviors) in the failing executions. Thus, developers could examine a number of unnecessary statements for fault localization.

2.3.2 Dynamic Slicing

Dynamic slicing approach utilizes an execution history of a given input while computing a slice [6, 8]. Specifically, the computed slice consists of control dependencies and data dependencies that indeed occur in a specific execution. Thus, dynamic slicing can further reduce the number of statements that need to be examined to localize faults, when compared to the static slicing approach. This is because dynamic slicing computes a slice that actually affects a slicing criterion (i.e., a variable that manifests a failure), whereas static slicing computes a slice that could affect a slicing criterion.

The primary disadvantage of dynamic slicing is the cost for the computation of dynamic slice. Even though several algorithms for dynamic slicing have been proposed [16, 81, 82], computing the precise dynamic slice could take a excessive time.

		Coverage of Test Cases (x, y)						Tarantula		Ochiai		Op2		FIESTA	
	int example (**int** x, **int** y){	TC_1 (5,1)	TC_2 (9,1)	TC_3 (9,0)	TC_4 (10,1)	TC_5 (1,2)	TC_6 (4,7)	Susp.	Rank	Susp.	Rank	Susp.	Rank	Susp.	Rank
s_1:	**int** ret = 0;	•	•	•	•	•	•	0.50	7	0.41	7	0.17	7	0.92	6
s_2:	**if** (x>y){	•	•	•	•	•	•	0.50	7	0.41	7	0.17	7	0.92	6
s_3:	x = x − 2; //should be 'x=x+2;'	•	•	•	•			0.63	3	0.50	3	0.50	3	0.95	2
s_4:	ret = ret + 1; }	•	•	•	•			0.63	3	0.50	3	0.50	3	0.95	2
s_5:	**else**{ y = y + 2; }					•	•	0.00	8	0.00	8	-0.33	8	0.36	8
s_6:	**if** (x<y+6){	•	•	•	•	•	•	0.50	7	0.41	7	0.17	7	0.92	6
s_7:	ret = ret + 2; }	•				•	•	0.71	1	0.58	1	0.67	1	0.90	7
s_8:	**return** ret; }	•	•	•	•	•	•	0.50	7	0.41	7	0.17	7	0.92	6
	Test Results	Fail	Pass	Pass	Pass	Pass	Pass								

Figure 2.1: Example of spectrum-based fault localization

2.3.3 Execution Slicing

To reduce the cost for computing a dynamic slice, the execution slicing approaches have been proposed by researchers [7, 56]. These approaches essentially utilize coverage information of a particular execution (e.g., a set of statements covered by an execution), called an execution slice. For instance, a set of statements covered by a failing execution can be examined by a developer for fault localization.

Agrawal et al. are one the first researchers who use an execution slice for fault localization purpose [7]. Given a failing and a passing test case, they compute 'dice', which is a set of statements that are only executed by a failing test case (not by a passing test case). Their approach reports the *dice* as likely faulty statements, because the dice is strictly correlated with failing executions. Renieres et al. [56] extend the Agrwl et al.'s approach. Their approach, called Nearest Neighborhood (NN), selects a passing test case that is most similar to the given failing test case in terms of code coverage. Then, the differences between the covered statements by the selected passing test case and the covered statements by the given failing test case are reported as likely faulty statements.

Although execution slicing-based approaches can decrease the search space for fault localization without requiring much computation cost, the precision of fault localization can decrease significantly. For example, a set of statements (e.g., dice) that is reported to a developer actually cannot contain faulty statements if all faulty statements are executed by both failing and passing test cases. It implies that a developer cannot find faults if her only searches for the statements reported by the fault localization technique.

Moreover, both the dynamic slicing-based approaches and execution slicing-based approaches still require a developer inspects a number of statements for fault localization (Section 2.4). For example, NN technique requires a developer to inspect 50% of PUT, on average [56]. Thus, the effectiveness of those approaches should be improved further to use them practically in the real world [54].

2.4 Spectrum-based Fault Localization

Program spectra characterizes an execution information of PUT. For instance, Executable Statement Hit Spectrum (ESHS) records which statements are executed by a test case (For more detailed information, see [28, 57, 71]).

The key idea of spectrum-based fault localization (SBFL) techniques is to utilize correlation between failing test cases and executed faulty entities. Specifically, they compute the suspiciousness of an entity (such as a statement or branch), which represents the degree of the possibility that an entity is faulty one, by analyzing differences between the program spectrum of passing executions and those of failing

executions. If the statement is used as the entity in SBFL (i.e., ESHS spectra is used), the suspiciousness of a statement s increases as the number of failing test cases that execute the statement s increases. Conversely, the suspiciousness of a statement s decreases as the number of passing test cases that execute the statement s increases. In other words, SBFL expects the correlation between failing executions and faulty statements is higher than the correlation between failing executions and correct statements. A representative SBFL technique called Tarantula [36] computes a suspiciousness of a statement s by using the following metric:

$$Susp_{Tarantula}(s) = \frac{\frac{|failing(s)|}{|T_f|}}{\frac{|failing(s)|}{|T_f|} + \frac{|passing(s)|}{|T_p|}}$$

where T_f is a set of failing test cases and $failing(s)$ is a set of failing test cases that execute s (T_p and $passing(s)$ are defined similarly).

Figure 2.1 shows an example of how the above metric localizes a fault. Let us assume that we have six test cases ($tc1$ to $tc6$) and statement s_3 is the faulty statement because it should be `x=x+2`. The executed statements by each test case (i.e., ESHS spectra) are marked with black dot in the figure. In the figure, $tc1$ is the only failing test case because `example()` with $tc1$ will return 3, while it will return 1 if there is no fault (i.e., if statement s_3 is `x=x+2`). All other test cases $tc2$ to $tc6$ are passing test cases because `example()` with these test cases will return a correct value (i.e., 1 with $tc2$ to $tc4$ and 2 with $tc5$ and $tc6$).

With the suspiciousness metric of Tarantula, the faulty statement (statement s_3) has $\frac{\frac{1}{1}}{\frac{1}{1}+\frac{3}{5}} = 0.63$ as a suspiciousness and marked as a rank 3. Thus, one can localizes the faulty statement after examining the three statements (s_7, s_4, and s_3) by following the order of statements in the ranking produced by Tarantula. Note that a developer should examine all statements executed by the failing test case `tc1` if the dynamic slicing approach is used for fault localization; Given the slicing criterion `ret` in statement s_8 that manifests failures, all statements executed by `tc1` are included into the computed slice. This is because all the statements affect the value of `ret` via control or data dependencies of `example()` in the failing execution (`tc1`).

The superior precision of Tarantula was empirically proved by Jones et al. [35]. They compared Tarantula technique with other representative fault localization techniques [7, 21, 56] that had been proposed. In their empirical evaluation on Siemens suite [29], Tarantula technique significantly outperforms other fault localization techniques - for instance, it ranks the faulty statement among the top 1% of executable statements for 14% of faults studied, which is three times more effective than the technique previously showed to be the most effective at fault localization on the Siemens suite. Since then, many researchers who were encouraged by the promising results have researched SBFL techniques to further improve the effectiveness. Some use multiple coverage entities [61], some generate certain test cases [9], some combine SBFL with other fault localization techniques [10, 11, 12, 25], and some utilize new suspiciousness metric for SBFL [4, 30, 47, 50]. For example, Ochiai [50] and Op2 [47] suspiciousness metrics are defined as follows:

$$Susp_{Ochiai}(s) = \frac{|failing(s)|}{\sqrt{|T_f| * (|failing(s)| + |passing(s)|)}}$$

$$Susp_{Op2}(s) = |failing(s)| - \frac{|passing(s)|}{|T_p| + 1}$$

Ochiai is generally known to outperform Tarantula, and Op2 is theoretically proved to outperform the other SBFL formulas proposed so far in single fault programs [74] (For more detailed information about SBFL formulas, see [47, 74]).

Although SBFL has received much attention, it has also been criticized for their impractical accuracy. Even if SBFL techniques are used, the number of statements that need to be examined until reaching the faulty statement is too many to use the techniques practically [54]. It is known the primary causes that decrease the effectiveness of SBFL techniques are (1) The negative effect of Coincidentally correct test cases (CCTs), which are test cases that pass despite executing the faulty statements, and (2) The coarse granularity of block level of SBFL.

Negative effect of coincidentally correct test cases Consider the Figure 2.1 again. In the Tarantula metric, statement s_7 ($ret = ret + 2$) has the highest suspiciousness (i.e., $\frac{\frac{1}{1}}{\frac{1}{1}+\frac{2}{5}} = 0.71$) and marked as rank 1, while the faulty statement has $\frac{\frac{1}{1}}{\frac{1}{1}+\frac{3}{5}} = 0.63$. This is because there are more passing test cases that execute statement s_3 (i.e., $tc2$, $tc3$, and $tc4$) than s_7 (i.e., $tc5$ and $tc6$). Since CCTs such as $tc2$ to $tc4$ increase $|passing(s)|$ in the denominator of the suspiciousness metrics (Tarantula and Ochiai) or in the negative term of the suspiciousness metric (Op2), the suspiciousness of the faulty statement becomes low. Thus, SBFL formulas like Tarantula, Ochiai, and Op2 become imprecise as there are more CCTs.

Coarse granularity of block level The program spectrum used by SBFL techniques is simply a combination of control flow of PUT and the results from test cases. Thus, all statements in the same basic block share the same spectra and, therefore, the same ranking. For example, in Figure 2.1, statement s_3 and s_4 are in the same basic block, thus they share same ranking. Therefore, the number of statements that need to be examined until reaching faulty statements will increase, as there are more statements in the same basic block.

2.5 Other Debugging Techniques

We will introduce other kinds of debugging techniques including fault localization techniques relevant to our approaches in Section 3.9 and Section 4.6, after describing the fundamentals of our approaches. Those include other kinds of SBFL techniques, delta debugging [21, 78], angelic debugging [19], IVMP [31], and so on.

Chapter 3. FIESTA: Fault Localization to Mitigate the Negative Effect of Coincidentally Correct Tests

This chapter presents our fault localization technique, called FIESTA, that mitigates the negative effect of Coincidentally Correct Test cases (CCTs). First, the chapter presents the intuitions of the proposed approaches, and then describe detailed techniques to mitigate the negative effects of CCTs. Second, the chapter presents empirical set-up and empirical evaluation results of FIESTA on the 12 subject programs. Third, the chapter presents several limitations of SBFL techniques including FIESTA, observed through the experiments. Fourth, the chapter discusses the effectiveness of FIESTA, and its practicality. Finally, the chapter compares our approaches with others that have attempted to solve the CCT problem.

3.1 Intuitions

The effectiveness of Spectrum-based fault localization (SBFL) techniques suffer from the existence of Coincidentally Correct Test cases (CCTs). As we showed in Section 2.4, the rank of faulty statement can be lower than correct statements because CCTs decrease the suspiciousness of faulty statement. A main cause of the negative effect of CCTs is that SBFL techniques *only consider execution result of each test case*, when computing suspiciousness of each statement. Previous SBFL formulas [30, 36, 47, 50] always decrease suspiciousnesses of statements if they are executed by the test cases that pass. However, the suspiciousnesses of statements executed by particular passing test cases, which execute the faulty statements, should be increased. For instance, the suspiciousness of statements executed by passing test cases, which execute same statements as a failing test case does, should be increased because they apparently execute the faulty statements. It implies that SBFL formulas should consider the passing test cases that are (likely) to execute faulty statements, when computing suspiciousness of each statement to improve the precision. Our new suspiciousness formula for SBFL considers such passing test cases when computing the suspiciousness (Section 3.2).

Another way to overcome the negative effect of CCTs can be to remove CCTs. Although it is generally not possible to remove CCTs because we do not know if a given passing test case executes a faulty statement, we can think of a way that could partially remove CCTs for a certain type of faults, i.e. faults in compound conditional statements. Consider a program that contains a compound conditional statement s_{comp} whose boolean condition (i.e., a predicate) consists of multiple clauses $c_1, c_2, ...c_n$ combined with logical operators. If one such clause c_m is faulty, s_{comp} becomes faulty and a passing test case tc_p that executes s_{comp} becomes a CCT, although tc_p may not execute c_m due to short-circuit evaluation. Consequently, tc_p decreases a suspiciousness score of s_{comp}, since tc_p covers s_{comp} but passes. We can mitigate this problem by decomposing a compound conditional statement s_{comp} into semantically equivalent nested atomic conditional statements (Section 3.3).

Based on the above two observations, we develop two techniques, which are called *fault weight based suspiciousness metric* and *debuggability transformation*, respectively. Our fault localization technique, Fault-weIght and atomizEd condition baSed localizaTion Algorithm (FIESTA), first transforms PUT to

the semantically equivalent one using the debuggability transformation technique to reduce the number of CCTs. It then uses the fault weight based suspiciousness metric, that can mitigate the negative effect of CCTs, in order to compute suspiciousness of each statement. Following sections describe the detailed approaches.

3.2 Fault Weight Based Suspiciousness Metric

3.2.1 Fault Weights on Test Cases

We define a *fault weight* w_f on a test case t, which indicates *likelihood* of t to execute a faulty statement. For a failing test case t_f, $w_f(t_f)$ is defined as 1. For a passing test case t, fault weight $w_f(t)$ is defined as follows:

$$w_f(t) = \frac{\Sigma_{t_f \in T_f} \frac{|stmt(t) \ \cap \ stmt(t_f)|}{|stmt(t_f)|}}{|T_f|}$$

where $stmt(t)$ is a set of statements executed by a test case t and T_f is a set of failing test cases. $w_f(t)$ represents a degree of overlap between the statements executed by t (i.e., $stmt(t)$) and the statements executed by a failing test case $t_f \in T_f$ (i.e., $stmt(t_f)$) on average over T_f. Thus, high $w_f(t)$ score indicates that t is likely to execute a faulty statement.

For the example of Figure 2.1 where $T_f = \{tc1\}$, we can compute the fault weights of the six test cases as follows:

- $w_f(tc1) = 1$, since $tc1$ is a failing test case.
- $w_f(tc2) = \frac{\frac{|stmt(tc2) \cap stmt(tc1)|}{|stmt(tc1)|}}{1} = \frac{\frac{6}{7}}{1} = 0.86$
- $w_f(tc3) = w_f(tc4) = w_f(tc2) = 0.86$, since $stmt(tc3) = stmt(tc4) = stmt(tc2)$.
- $w_f(tc5) = \frac{\frac{|stmt(tc5) \cap stmt(tc1)|}{|stmt(tc1)|}}{1} = \frac{\frac{5}{7}}{1} = 0.71$
- $w_f(tc6) = w_f(tc5) = 0.71$, since $stmt(tc6) = stmt(tc5)$

The fault weights (i.e., 0.86) of CCTs (i.e., $tc2$, $tc3$, and $tc4$) are higher than the fault weights (i.e., 0.71) of TCTs (Truly Correct Test cases) (i.e., $tc5$, $tc6$), which are passing test cases that do not execute a faulty statement. If $w_f(t_p)$ is high for a passing test case t_p, t_p is likely a CCT (see Figure 3.2) and we can make a precise suspiciousness metric that gives high suspiciousness to the statements executed by such t_p.

3.2.2 Suspiciousness Metric of FIESTA

We define a new suspiciousness metric $Susp_{FIESTA}$ on a statement s using the fault weights on test cases as follows:

$$Susp_{FIESTA}(s) = (\frac{|failing(s)|}{|T_f|} + \frac{\Sigma_{t \in test(s)} w_f(t)}{|test(s)|})/2$$

where $test(s)$ is a set of test cases that execute s (i.e., $test(s) = passing(s) \cup failing(s)$).

```
/* Original Program P */
 s1:   if (x>0 && y==0 && z<0){  // 'z<0' should be 'z==0'
 s2:     statement;
 s3:   }

/* Transformed Program P' */
s11:   if (x>0){
s12:     if (y == 0){
s13:       if (z < 0){                  // 'z<0' should be 'z==0'
s14:         statement;
s15:   }}}
```

Figure 3.1: Example of P and P' that has been transformed from P by FIESTA

The left term, $\frac{|failing(s)|}{|T_f|}$, increases the suspiciousness, if s is executed by more failing test cases. [1] The right term, $\frac{\Sigma_{t \in test(s)} w_f(t)}{|test(s)|}$, increases the suspiciousness, if s is executed by test cases with higher fault weights. Finally, to normalize a metric value within a range 0 to 1, the sum of the left term and the right term is divided by 2.

For example, with the example of Figure 2.1, the new fault localization metric computes the suspiciousness of statement s_3 as 0.95 and marks s_3 (and statement s_4 together) as the first rank as follows:

$$(\frac{1}{1} + \frac{w_f(tc1) + w_f(tc2) + w_f(tc3) + w_f(tc4)}{4})/2 = 0.95$$

Note that the suspiciousness metric of FIESTA is more precise than those of Tarantula, Ochiai, and Op2 for this example because FIESTA utilizes the fault weights to increase suspiciousness scores of the statements executed by CCTs.

3.3 Debuggability Transformation of PUT

3.3.1 Overview

FIESTA transforms a target program P into the semantically equivalent program P' by converting a compound conditional statement into nested conditional statements with atomic boolean conditions (i.e., conditions without boolean operators) [2]. For example, FIESTA transforms a compound conditional statement at statement s_1 of P in Figure 3.1 into the semantically equivalent nested conditional statements in P' (statements $s_{11} \sim s_{13}$).

Suppose that the last atomic clause of statement s_1 in P is a fault (e.g., it should be `z==0`, not `z<0`) and a test case tc_1(`x=0,y=0,z=0`) that executes s_1 is a CCT for P. Note that tc_1 is *not* a CCT for P' since tc_1 does not execute the corresponding faulty statement (line 13) of P' (see Section 3.3.2). CCTs that execute statement s_1 of P are *not* CCTs for P' unless they satisfy `x>0 && y==0`. Thus, through the transformation of a target program P to P', such CCTs become truly correct test cases (TCTs) (i.e., passing test cases that do not execute a faulty statement) and the precision of coverage-based fault localization can be improved on P' due to the decreased number of CCTs.

[1] The left term may make FIESTA localize a fault less precisely on a target program with multiple faults spanning on multiple lines than one with a single line fault, which occurs commonly to most SBFL techniques. However, Digiuseppe et al. [22] showed that SBFL techniques are still effective to target programs with multiple faults (i.e., in the situation where different faults interfere with each other's localizability) as the number of the faults increases. In addition, there are several techniques (i.e., Liu et al. [41], Zheng et al. [83], and Jones et al. [34]) to cluster test cases targeting an individual fault to reduce interference among multiple faults.

[2] FIESTA also transforms `exp?v1:v2` in P to `if(exp) tmp=v1; else tmp=v2;` in P'.

Once the suspiciousness scores on the statements of P' are obtained, these scores are mapped to the statements of P (see Section 3.3.2). Finally, the user reviews the statements of P to identify a faulty statement as usual.

3.3.2 Mapping of Statements between P and P'

FIESTA utilizes CIL (C Intermediate Language) [49] to transform a target C program P to P' and CIL generates basic mapping information between statements of P and P'. Based on this mapping information, FIESTA maps a faulty statement in P to statement(s) in P'. In addition, FIESTA maps the suspiciousness scores on the statements in P' to the original statements in P.

Mapping Faulty Statements of P to P'

To measure how many CCTs are reduced through the transformation, we map a faulty compound conditional statement of P to atomic conditional statements of P'. There are four cases to consider for the mapping as follows:

1. *When a clause of a compound condition is faulty (e.g.,* `z<0` *is faulty in* `if(x>0 && y==0 && z<0)`*, since it should be* `z==0`*):*

   Statement(s) of P' that correspond to a faulty clause (i.e., `if(z<0)` in P') are marked as faulty statements.

2. *When a boolean operator is faulty (e.g.,* `if(x>0 && y==0) {s1;}` *should be* `if(x>0 || y==0) {s1;}`*):*

   Statement(s) of P' that correspond to the *all operands* of the faulty operator (e.g., `if(x>0)` and `if(y==0)` in P') are marked as faulty statements. This is because each of those statements that correspond to the operands can trigger errors.

   For example, with tc_2 (`x=1`,`y=1`), the correct program (i.e., `if(x>0 || y==0) {s1;}`) will execute `s1`, but the faulty transformed program (i.e., `if(x>0) { if(y==0) {s1;}}`) will not execute `s1` due to `if(y==0)`. Similarly, with tc_3 (`x=0`, `y=0`), the correct program will execute `s1`, but the faulty transformed program will *not* execute `s1` due to `if(x>0)`.

3. *A new clause is added (e.g.,* `w!=0` *is conjunctively added):*

   Statement(s) of P' that correspond to the newly added clause (e.g., `if(w!=0)` in P') are marked as faulty statements.

4. *A clause is omitted (e.g.,* `if(x<0 /*&& y<0 omitted*/) {s1;}`*):*

   Statement(s) of P' that correspond to the statements that are control dependent on the omitted clause in P (i.e., `y<0` in the example) are marked as faulty (i.e., `s1` in the example), since such statements can trigger errors.

Mapping Suspiciousness Scores on the Statements of P' to P

Although we obtain suspiciousness scores on the statements of P', we have to examine statements of P in the descending order of their suspiciousness. This is because we would like to identify a fault in the original program eventually. In addition, P' can be larger than P through the transformation, which may cause a user to examine more statements even when the % of the executed statements to examine

to find a faulty statement for P' is smaller than the % of the executed statements to examine for P. Thus, FIESTA maps the suspiciousness scores on the statements of P' to the statements of P.

Suppose that a compound conditional statement s_{comp} of P is transformed into nested conditional statements with atomic conditions $s'_1, s'_2, ...s'_n$. Then, the suspiciousness score of s_{comp} is defined as the score of $s'_m (1 \leq m \leq n)$ that has the *highest* suspiciousness score among the scores of all $s'_1, s'_2, ...s'_n$. In addition, FIESTA reports s'_m to a user to help the user examine suspicious statements to localize a fault. The suspiciousness scores of the statements of P' that are not expanded from a compound conditional statement in P are directly mapped to their original statements in P.

3.4 Overall Procedure

FIESTA constructs the ranking of statements of a target program P whose execution with some test cases results in failures, by following below procedures. The generated ranking of statements will be consumed by a developer for fault localization.

First, FIESTA transforms a target program P into the semantically equivalent program P' by converting each compound conditional statement into nested conditional statements with atomic conditions (debuggability transformation).

Second, FIESTA executes the transformed program P' with a given test suite to get Executable Statement Hit Spectra (ESHS) of each test case.

Third, FIESTA computes suspiciousness of each statement using the FIESTA suspiciousness metric based on the ESHS of each test case.

Lastly, FIESTA maps suspiciousness of each statement in P' to P based on the rule defined in Section 3.3.2, and ranks statements in P according to their suspiciousness.

3.5 Empirical Study Setup

We design the following four research questions to study effectiveness of the transformation technique and the fault weight metric as well as the precision of FIESTA:

RQ1: *How effective is the transformation of a target program, in terms of a reduced number of CCTs?*

We measure both the number of CCTs for a target program P and the number of CCTs for the transformed target program P'.

RQ2: *How accurate is the fault weight metric of FIESTA on test cases, in terms of the probability of a passing test case with a high fault weight to be a CCT?*

We measure the number of passing test case whose fault weight is larger than 0.9 to be a CCT over all passing test cases of target programs. Also we observe trend between fault weight and

RQ3: *How precise is FIESTA, specifically compared to state-of-art SBFL techniques (i.e., Tarantula, Ochiai, and Op2) in terms of the % of executed statements examined to localize a fault?*

We measure the % of executed statements examined to localize a fault as a measurement of the precision. The precision of the fault localization technique will inversely correlate with the percentage.

RQ4: *How effective is the transformation of a target program, in terms of improved precision of FIESTA?*

We measure how much the debuggability transformation improves the precision of FIESTA, by comparing the precision with or without the debuggability transformation.

To answer these research questions, we have performed a series of experiments by applying Tarantula, Ochiai, Op2, and FIESTA to the 12 subject programs. We will describe the detail of the experiment setup in the following subsections.

3.5.1 Subject Programs

As objects of the empirical study, we use the seven SIEMENS programs (419.1 LOC on average) and five non-trivial real-world programs (`flex` 2.4.7, `grep` 2.2, and `gzip` 1.1.2, `sed` 1.18, and `space`) in the SIR benchmark suite [23] (10641.2 LOC on average). Among the faulty versions of those programs, we excluded faulty versions for which no test case or only one test case fails. Table 4.1 describes the subject programs and the number of their faulty versions on which we conducted fault localization experiments. The subject programs have 154 faulty versions each of which has a single fault (only one line is different from the non-faulty version) and 25 faulty versions each of which has multiple faults (more than one line are different from the non-faulty version). In addition, we used all test cases in the SIR benchmark except the ones that crash a subject program since `gcov` [1] we used to measure coverage information of each test case cannot measure coverage information if a target program crashes.

3.5.2 Experiment Setup

We implemented Tarantula, Ochiai, Op2, and FIESTA in 2600 lines of C++ code with CIL 1.3.7. The subject programs are compiled and instrumented in statement-level by using `gcc`. After each execution of the instrumented object program finishes, we use `gcov` to get the statement coverage of the program execution on a given test case. Our tools parse the statement coverage reported from `gcov` for each test case, and construct a table that shows which statements are covered by which test cases (see Figure 2.1). In our experiments, we consider only executed statements that can be tracked by `gcov`. Each executed statement is given a suspiciousness according to the suspiciousness metrics of Tarantula,

Table 3.1: Subject programs, their sizes in Lines Of Code (LOC), and the number of test cases

Subject program	LOC	# of faulty ver. used	# of provided test cases	Description
print_tokens	563	7	4130	lexical analyzer
print_tokens2	510	9	4115	lexical analyzer
replace	563	30	5542	pattern replacement
schedule	412	5	2650	priority scheduler
schedule2	307	9	2710	priority scheduler
tcas	173	40	1608	altitude separation
tot_info	406	23	1052	information measure
flex	12423	16	567	lexical analyzer generator
grep	12653	4	809	pattern matcher
gzip	6576	5	214	file compressor
sed	11990	3	360	stream editor
space	9564	28	13585	ADL interpreter
Average	4678.33	14.92	3111.83	

Ochiai, Op2, and FIESTA and ranked in a descending order of the suspiciousness scores. Statements with the same suspiciousness are assigned the lowest rank the statements can have(See Figure 2.1). This is because we assume that the developer must examine all of the statements with the same suspiciousness to find faulty statements. All experiments were performed on 5 machines that are equipped with Intel i5 3.6 Ghz CPUs and 8 Gigabyte of memory and run Debian 6.05.

3.5.3 Threats to Validity

The primary threat to external validity for our study involves the representativeness of our object programs since we have examined only 12 C programs. However, since the subject programs include various faulty versions of five real-world subject programs with test suites to support controlled experimentation and the subject programs are widely used for fault localization research, we believe that this threat is limited. The second threat involves the representativeness of the SBFL techniques that we applied (i.e., Tarantula, Ochiai, Op2, and FIESTA). However, since Tarantula and Ochiai are considered as representative SBFL techniques in literature, and Op2 is theoretically proven to outperform other SBFL formulas proposed, in single fault programs [74]. Thus, we believe that this threat is also limited. The third possible threat is the assumption that we already have many test cases when applying SBFL techniques to a target program. However, since we can generate many test cases using automated test generation techniques (i.e., random testing, concolic testing, or test generation strategies for fault localization [9, 25, 59]), this threat is limited too. A primary threat to internal validity is possible faults in the tools that implement Tarantula, Ochiai, Op2, and FIESTA. We controlled this threat through extensive testing of our tool.

3.6 Result of The Experiments

We have applied our implementation of Tarantula, Ochiai, Op2, and FIESTA to the 179 versions of the 12 programs (the whole experiments took around three hours by running 5 machines). From the

Table 3.2: Reduced number of CCTs through the debuggability transformation

Subject program	# of ver. with faulty comp. stmt.	# of faulty comp. stmt. per version	# of lines of expanded faulty stmt.	# of CCTs in P	# of CCTs in P'	Reduced ratio (%)
`ptok2`	3	1.00	2.67	1947.33	1897.33	2.57
`replace`	11	1.00	5.27	1647.55	1086.09	34.08
`schedule2`	4	1.00	4.00	2618.75	2166.25	17.28
`tcas`	20	1.20	5.38	1049.37	716.68	31.70
`tot_info`	1	1.00	2.00	844.00	806.00	4.50
`flex`	3	1.00	2.67	330.67	58.67	82.26
Average	7.0	1.03	3.66	1406.28	1121.84	28.73

data obtained from the series of experiments, we can answer the four research questions in the following subsections.

3.6.1 Regarding RQ1: Reduced Number of CCTs through the Debuggability Transformation

Table 3.2 describes statistics on the reduced number of CCTs for the subject program versions that have faulty compound conditional statements through the transformation (see Section 3.3). [3] Among the total 179 versions of the subject programs, only 42 (=3+11+4+20+1+3) versions have faulty compound conditional statements (see the second column of Table 3.2). For the 42 versions of the subject programs that have faulty compound conditional statements, the number of CCTs decreases 28.73% on average (see the last column of Table 3.2). For example, three versions among the 16 faulty versions of `flex` have 1.00 faulty compound conditional statement each on average (see the first to the third columns of the seventh row of Table 3.2). In addition, each such compound faulty statement is transformed into 2.67 statements on average (see the fourth column of the seventh row of Table 3.2) and the number of CCTs decreases from 330.67 for P to 58.67 for P' on average (i.e., 82% of CCTs are reduced) (see the fifth to the seventh columns of the table). Therefore, we can conclude that the debuggability transformation technique effectively reduces the number of CCTs for faulty compound conditional statements.

Note that bug studies in literature show that incorrect conditional statements were often the causes of bugs. For example, the causes of 22% of the bugs in the operating system at IBM [20] and 25% of bugs in the 12 open-source programs [24] resided in the incorrect conditional statements. It implies that FIESTA can precisely localize faulty statements in real-world programs, due to the debuggability transformation technique that effectively reduces the number of CCTs for the incorrect compound conditional statements.

3.6.2 Regarding RQ2: Accuracy of the Fault Weight Metric on Test Cases

Table 3.3 describes statistics on the test cases used to localize a fault in the subject programs. The number of CCTs (i.e., 1111.5) is similar to TCTs (i.e., 1717.3) on average (see the third column and the fifth column of the last row of the table). [4] Average fault weight of CCTs (i.e., 0.86) is higher than that of TCTs (i.e., 0.66) (see the fourth column and the sixth column of the last row of the table).

[3] For a program version which does not contain a faulty compound conditional statement, the number of CCTs does not change for P'.

[4] Note that the numbers of CCTs may be different for different versions of a subject program P due to the different faulty lines in the different versions of P.

Table 3.3: Fault weights of test cases for the subject programs

Subject programs	Failing test cases	Passing test cases				Total test cases used	
		CCT		TCT			
	Avg. # of test cases	Avg. # of test cases	Avg. fault weight	Avg. # of test cases	Avg. fault weight	Avg. # of test cases	Avg. fault weight
ptok	69.1	1960.1	0.75	2100.7	0.69	4130.0	0.72
ptok2	229.3	1187.9	0.83	2697.8	0.76	4115.0	0.80
replace	100.7	2218.8	0.79	3222.5	0.65	5542.0	0.72
schedule	142.8	1431.2	0.96	1069.0	0.64	2643.0	0.83
schedule2	29.0	2518.9	0.90	153.6	0.50	2701.4	0.87
tcas	40.6	852.6	0.90	714.8	0.62	1608.0	0.78
tot_info	83.7	680.1	0.85	288.1	0.50	1052.0	0.77
flex	255.7	100.1	0.83	183.2	0.73	538.9	0.88
grep	241.5	36.0	0.87	486.8	0.67	764.3	0.78
gzip	47.6	37.4	0.96	128.0	0.81	213.0	0.91
sed	48.7	13.0	0.86	234.0	0.72	295.7	0.77
space	1954.2	2302.1	0.83	9328.6	0.67	13585.0	0.74
Average	270.3	1111.5	0.86	1717.3	0.66	3099.0	0.80

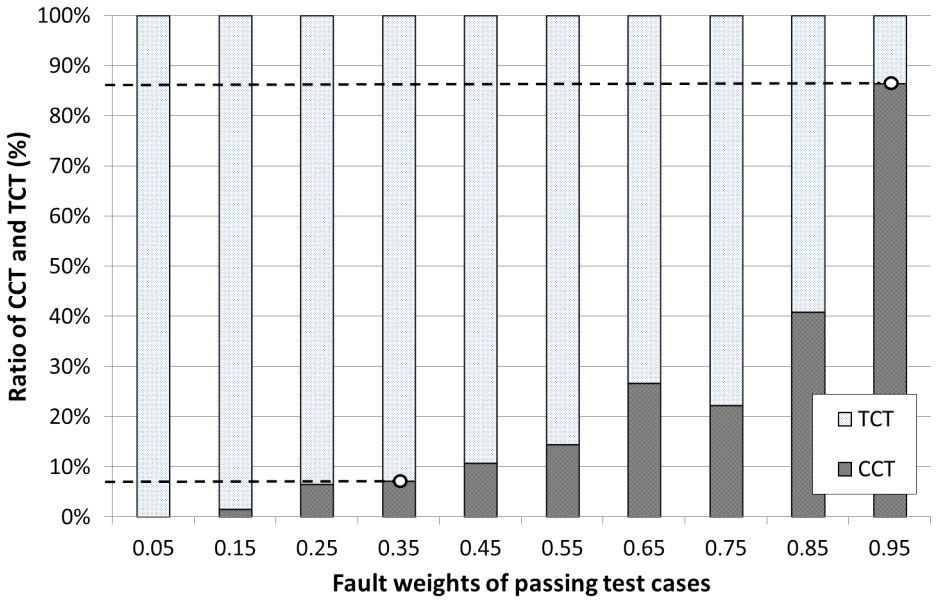

Figure 3.2: Ratios of CCTs and TCTs for different fault weights

Figure 3.2 shows that the ratios of CCTs and TCTs for the 179 versions of the 12 subject programs according to their fault weight values. Figure 3.2 shows that passing test cases with high weights are more likely CCTs. For example, a set of passing test cases whose weights are between 0.9 and 1 (mean value 0.95) consists of 86.4% CCTs and 13.6% TCTs (see the small circle at the top right part of Figure 3.2). In other words, the probability of a passing test case with weight larger than 0.9 to be a CCT is 86.4%. In contrast, the probability of a passing test case with a fault weight between 0.3 and 0.4 (mean value 0.35) to be a CCT is only 7.1% (see the small circle at the bottom of Figure 3.2). Therefore, we can conclude that the proposed fault weight metric is accurate enough to distinguish CCTs and TCTs approximately. Consequently, FIESTA that utilizes the fault weight metric can improve the precision of fault localization by reducing the negative effect of CCTs.

Table 3.4: % of executed statements examined to localize a fault in the subject programs (i.e., precision)

Subject program	% of executed stmts examined			
	Tarantula	Ochiai	Op2	FIESTA
`print_tokens`	27.45	19.02	**10.41**	11.20
`print_tokens2`	12.88	7.47	**1.62**	2.31
`replace`	9.19	6.98	**5.44**	7.74
`schedule`	5.95	**3.51**	17.17	10.45
`schedule2`	52.85	46.83	40.48	**28.55**
`tcas`	28.31	28.62	27.14	**21.17**
`tot_info`	27.30	21.31	**13.17**	16.11
`flex`	29.55	**12.51**	13.54	12.84
`grep`	21.71	1.68	**1.23**	**1.23**
`gzip`	17.15	7.91	7.91	**7.49**
`sed`	4.94	1.32	**1.29**	**1.29**
`space`	5.83	**2.80**	4.90	3.70
Average	20.26	13.33	12.03	**10.34**

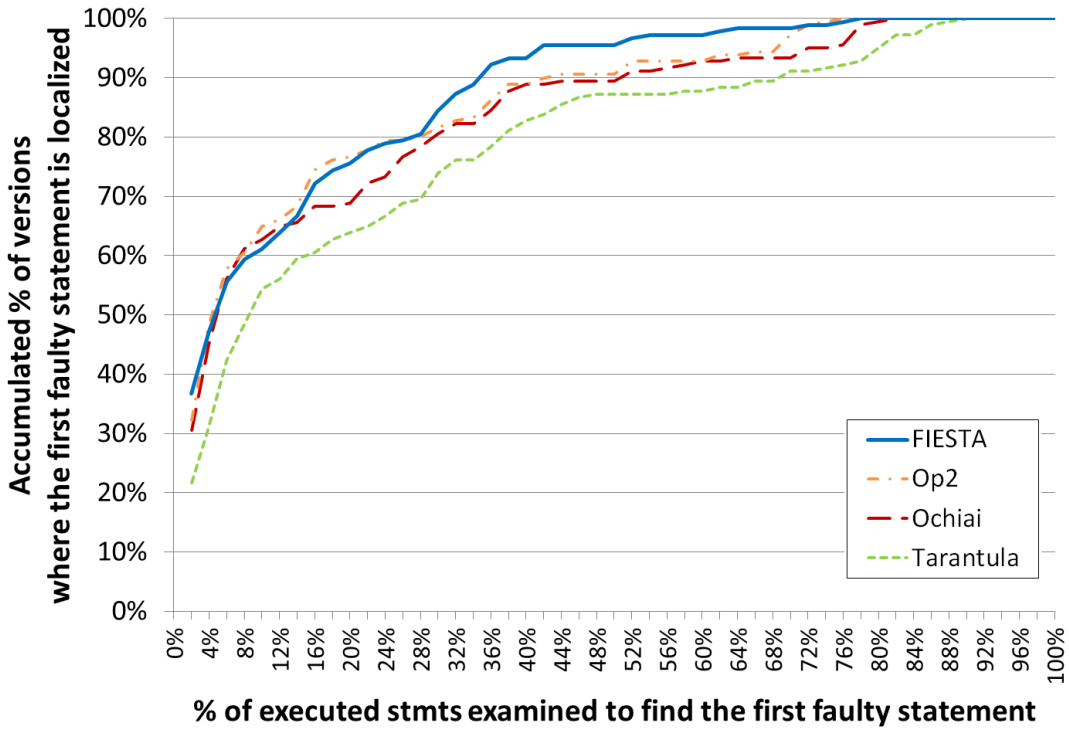

Figure 3.3: Accumulated % of subject versions whose faults are localized after examining a certain amount of target statements

3.6.3 Regarding RQ3: Precision of FIESTA

Table 3.4 enumerates the precisions of fault localization on the subject programs by using Tarantula, Ochiai, Op2, and FIESTA. We mark the most precise results in bold font. For the programs, FIESTA localizes a fault after reviewing 10.34% of executed statements on average (the last column of the last row of the table).

The comparison results with Tarantula, Ochiai, and Op2 are as follows. FIESTA is 49% ($=\frac{20.26-10.34}{20.26}$),

Table 3.5: Effect of the transformation technique on the precision of FIESTA

Subject programs	% of executed stmts examined		Relative Imprv.(%)
	w/o Trans.	w/ Trans.	
p_tokens	11.20	11.20	0.00
p_tokens2	2.41	2.31	4.29
replace	10.88	7.74	28.82
schedule	10.32	10.45	-1.27
schedule2	43.01	28.55	33.62
tcas	27.29	21.17	22.44
tot_info	15.76	16.11	-2.22
flex	13.75	12.84	6.67
grep	1.23	1.23	0.00
gzip	7.51	7.49	0.30
sed	1.29	1.29	0.00
space	3.70	3.70	0.00
Average	12.36	10.34	7.72

22% ($=\frac{13.33-10.34}{13.33}$), and 14% ($=\frac{12.03-10.34}{12.03}$) relatively more precise than Tarantula, Ochiai, and Op2, on average, respectively (see the last row of Table 3.4). In addition, FIESTA localizes a fault more precisely in all subject programs except `schedule` than Tarantula. With the exception of `replace`, `schedule`, `flex`, and `space`, FIESTA localizes a fault more precisely than Ochiai. Compared to Op2, FIESTA localizes a fault more precisely in the six programs (i.e., `schedule`, `schedule2`, `tcas`, `flex`, `gzip`, and `space`) and localizes a fault with the same precision in `grep` and `sed`. Furthermore, in the five real-world programs, FIESTA localizes a fault more or equally precisely compared to Op2.

Also, Figure 3.3 shows that FIESTA is more precise than Tarantula, Ochiai, and Op2 in a different aspect. Figure 3.3 illustrates that faults of how many subject program versions (y axis) can be detected by examining a given % of subject program executed statements (x axis). As showed in Figure 3.3, FIESTA always detects a fault in more subject versions than Tarantula does after examining the same number of target statements. FIESTA always detects a fault in more subject versions than Ochiai and Op2 after examining the same number of target statements, except when % of executed statements examined is between 8-12% and 4-20% for Ochiai and Op2 respectively. For example, by examining 2% of executed statements, FIESTA can find faults in 37% of all 179 subject program versions (i.e., 67 (=179×37%) versions) while Tarantula, Ochiai, and Op2 can find faults in 22% (40 versions), 31% (56 versions), and 33% (59 versions) of the subject program versions respectively.

3.6.4 Regarding RQ4: Improved Precision due to the Debuggability Transformation

Table 3.5 shows how much the debuggability transformation improves the precision of FIESTA. The table shows that the debuggability transformation relatively increases the precision FIESTA by 7.72% on average. For example, for `replace`, FIESTA without the transformation finds a fault after reviewing 10.88% of the executed statements while FIESTA (with the transformation) does after reviewing 7.74% of the executed statements, which is 28.82% ($=\frac{10.88-7.74}{10.88}$) relatively more precise result (see the fourth row of Table 3.5).

Table 3.6: Effect of the transformation technique on the precision of FIESTA for versions containing faulty compound conditional statements

Subject program	% of executed stmts examined		Relative Imprv.(%)
	w/o Trans.	w/ Trans.	
p_tokens2	1.88	1.41	25.00
replace	8.43	2.57	69.49
schedule2	65.41	31.08	52.48
tcas	37.94	24.05	36.62
tot_info	14.41	10.17	29.41
flex	7.14	2.14	69.97
Average	22.53	11.90	47.16

The transformation improves the precision much for the subject versions that contain faulty compound conditional statements. Table 3.6 shows the improved precision on such versions only (i.e., 42 out of 179 versions (see the second column of Table 3.3)). For such versions of the subject programs, the debuggability transformation significantly increases the precision of FIESTA by 47.16%. [5]

Therefore, if we can generalize the experiment results, we can conclude that the debuggability transformation technique can improve the precision of FIESTA.

3.7 Limitations

Although FIESTA detected a fault by examining 10.34% of the executed statements on average (see Table 3.4), there are 8 versions out of the total 179 versions of the subject programs for which FIESTA examined more than 50% of executed statements to localize a fault. We can classify these 8 versions into the following four categories according to the reasons for the low precision:

- A large number of statements covered together (one version)
- Correct statements executed by only failing test cases and CCTs (one version)
- A fault in a large basic block (three versions)
- Subject program specific reasons (three versions)

Through the detailed analysis of these versions, we found that the first three categories are common ones for most SBFL techniques, not specific for FIESTA. For example, the precision of FIESTA for these versions is 62.99% on average and the precision of Tarantula, Ochiai, and Op2 for these versions is 74.40%, 67.26%, and 53.66% on average respectively.

3.7.1 Large Number of Test Cases that Cover the Same Statements

The precision of FIESTA for flex version 8 was 52.11%. A main reason for this imprecise result is that more than half of executed statements are covered by the same set of test cases. Specifically, 396 statements (52.11% of executed statements) including the faulty statements are covered together by each of the 553 test cases (out of total 567 test cases), so that those 396 statements have the same suspiciousness

[5] For the 137 versions that have no faulty compound conditional statements, the transformation relatively decreases the precision of FIESTA by 0.76%.

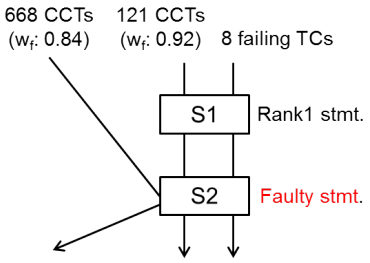

Figure 3.4: Exceptional case where a correct statement $S1$ is executed by only failing test cases and CCTs

scores. Although the suspiciousness of the faulty statement is the highest, % of code to examine is more than 50%, since the 396 statements that have the same suspiciousness should be examined altogether. This problem can be solved by constructing test cases to execute different execution paths.

3.7.2 Correct Statements Executed by Only Failing Test Cases and CCTs

The precision of FIESTA for `tot_info` version 10 was 51.75%. An important reason is that this version has statements that are executed by only failing test cases and CCTs. Figure 3.4 shows the case of `tot_info` version 10. $S1$ is a correct statement that has a higher rank than a faulty statement $S2$ for the following reason. We found that all 8 failing test cases execute both $S1$ and $S2$, and there are 789 CCTs that execute $S2$ (121 CCTs execute both $S1$ and $S2$ and 668 CCTs execute $S2$ only) and $passing(S1) \subset passing(S2)$ (i.e., $S1$ is *not* executed by TCTs). In this situation, the 668 CCTs have lower fault weights on average (i.e., 0.84) than the 121 CCTs (i.e., 0.92), because the 668 CCTs execute a smaller number of statements that are executed by the failing test cases than the 121 CCTs execute. Thus, $S1$ has a higher suspiciousness than $S2$.

Note that this case is exceptional and problematic for most SBFL techniques such as Tarantula, Ochiai, and Op2. For example, the precision of Tarantula, Ochiai, and Op2 for this version are 60.53%, 58.77%, and 51.75%, respectively.

3.7.3 Fault in a Large Basic Block

`tcas` versions 3, 12, and 34 have a fault in an initial basic block of `main()` function. This basic block consists of 26 statements, which are more than 40% of the executed statements of `tcas` (i.e., these 26 statements are always executed together). Although the suspiciousness of the fault is 1 (i.e., the highest score), % of executed statements to examine is more than at least 40%, since the 26 statements in the same basic block should be examined together.

3.8 Discussion

FIESTA improves the precision of spectrum-based fault localization by reducing the negative effects of CCTs based on the debuggability transformation and the fault weight on test cases. Through the experiments on the 12 programs including five real-world programs of large sizes, we have demonstrated that FIESTA is 49%, 22%, and 14% relatively more precise than Tarantula, Ochiai, and Op2 on average, respectively. In addition, compared with Tarantula, Ochiai, and Op2, FIESTA localizes a fault equally or more precisely on 11, 8, and 8 subject programs out of the 12 subject programs on average, respectively.

Although FIESTA outperforms the state-of-art SBFL techniques, the effectiveness of fault localization techniques should be improved further to use fault localization techniques practically. For example, a developer with FIESTA should examine 1000 LOC to find a faulty statement if 10000 LOC are executed (FIESTA has ranked a faulty statement among the top 10.34% of executed statements on our subject programs (Section 3.6.3)).

However, the further improvement of the effectiveness is challenging if we only utilize SBFL. As SBFL techniques utilize program spectrum, they have the limitations: the negative effect of CCTs and the block level granularity (Section 2.4). Thus, other kinds of approaches that do not strongly depend on the program spectrum for fault localization should be developed. In the next chapter, we present a novel fault localization, which is free from the negative effect of CCTs and the block level granularity, thus improves the precision significantly.

3.9 Related Work

3.9.1 Techniques to Solve CCT Problems

Masri et al. [46] present an approach that removes CCTs by using the k-means clustering algorithm [44, 64]. The technique selects a set of suspicious statements (calling them CCE) that are executed by all failing test cases and a certain number of passing test cases (the number is given by a user as a threshold). The technique then clusters test cases into two groups based on the similarity of the executed statements of the test cases to the CCEs and the group that is more closely related to the CCEs is removed.

A coverage refinement approach adjusts program coverage to strengthen correlation between faults and failing test runs [67]. The work defines *context patterns* which are control and data flows before and after the execution of the faulty statements. They define context patterns according to fault types (e.g., missing assignment), and use the context patterns for adjusting coverage of test cases to reduce negative effect of CCTs. However, this approach requires human knowledge on the types of faults to define context patterns. In contrast, FIESTA does not require the types of program faults to be known, but still filters out negative effect of CCTs through the debuggability transformation (see Section 3.3) and the fault weights on test cases (see Section 3.2).

The motivation of the fault weight metric of FIESTA is similar to the related work described in this section, in a sense that all of these techniques try to estimate *distance* between passing test runs and failing test runs to localize a fault precisely. However, distance metrics utilized vary depending on the techniques (fault weight for FIESTA, and CCE distance for Masri et al. [46], Ulam distance for Renieres et al. [56], etc.). Furthermore, although all related work mentioned in this section may suffer false positives or false negatives to recognize CCTs, the debuggability transformation of FIESTA reduces certain number of CCTs without false positives nor false negatives, which can increase the precision of SBFL technique.

Bandyopadhyay [13] assigns a weight to a passing test case based on the CC-proximity [43] to improve the precision of SBFL. Although the idea of using weights on test cases is similar to FIESTA, the rationale behind the FIESTA metric is different from the Banyopadhyay et al.'s approach. In their approach, the weight on each passing test case is not to represent *likelihood* that a test case executes faulty statements. The purpose of their weight metric is to measure the *importance* of each passing test case. They argue that passing test cases that have similar coverage to the coverage of failing ones are

important than other passing ones, since the differences between the statements covered by those passing test cases and failing test cases may contain the faults [56]. In addition, as they replace $|passing(s)|$ term in Ochiai (see Section 2.4) with the average weight of passing test cases that execute statement s, the variant Ochiai formula can suffer from the negative effect of CCTs. In the variant formula, if there are many CCTs with high weights, the suspiciousness of faulty statements will decrease since the value of denominator increases, thus decreasing the precision of fault localization [6]. In contrast, in FIESTA suspiciousness formula, if there are many CCTs with high weights, the suspiciousness of faulty statement will increase, thus increasing the precision of fault localization.

3.9.2 Techniques to Utilize Predicates as a Target Program Spectrum

There have been several fault localization techniques that utilize predicates in the program code and additional predicates on program execution information. For example, Liblit et al. [39] and Liu et al. [42] instrument/transform a target program to record how program predicates (and additional predicates such as if a return value of a function is equal to zero) are evaluated and rank suspicious predicates. In other words, they utilize the numbers of failing tests and passing tests where a predicate is true (or false) to localize a fault.

These techniques can also be considered to perform a debuggability transformation to extract useful execution information for fault localization. The debuggability transformation of FIESTA, however, improves the precision of fault localization *actively* by reducing the number of CCTs explicitly (see Sections 3.3 and 3.6.1) while the above techniques improve the precision passively by observing more execution information. In addition, those techniques handle a predicate consisting of multiple clauses as a whole (although the effect of compound predicates were briefly discussed in Abreu et al. [3]). In contrast, FIESTA targets each clause of a predicate individually through the debuggability transformation.

[6] To reduce the negative effect of CCTs, the variant Ochiai formula requires user given thresholds, which are not required in FIESTA

Chapter 4. MUSE: Fault Localization by Mutating Faulty Programs

This chapter presents a novel fault localization technique, which utilizes mutation analysis [18]. First, this chapter describes intuitions of the proposed technique, and then describe detailed approaches. Second, the chapter presents empirical set-up and empirical evaluation results of the proposed technique on 5 real-world programs. Third, the chapter presents discussions about the effectiveness of the proposed technique. Finally, the chapter presents related works, and compares the proposed technique with other approaches.

4.1 Intuitions

Mutation testing [18] evaluates the adequacy of a test suite based on its ability to detect artificially injected faults, i.e. syntactic *mutations* of the original program. The more of the injected faults are *killed* (i.e. detected) by the test suite, the better the test suite is believed to be at detecting unknown, actual faults.

For the fault localization, we focus on what happens when we mutate an already faulty program and, particularly, the faulty program statement. Intuitively, since a faulty program can be repaired by modifying faulty statements, mutating (i.e., modifying) faulty statements will make more failed test cases pass than mutating correct statements. In contrast, mutating correct statements will make more passed test cases fail than mutating faulty statements. This is because mutating correct statements introduces new faulty statements in addition to the existing faulty statements in Program Under Test (PUT).

Consider a faulty program P whose execution with some test cases results in failures. We propose to mutate P knowing that it already contains at least one fault. Let m_f be a mutant of P that mutates the faulty statement, and m_c one that mutates a correct statement. MUSE depends on the following two conjectures.

Conjecture 1: test cases that used to fail on P are more likely to pass on m_f than on m_c. The first conjecture is based on the observation that m_f can only be one of the following three cases:

1. **Equivalent mutant** (i.e. mutants that syntactically change the program but not semantically), in which case the faulty statement remains faulty. Tests that failed on P should still fail on m_f.

2. **Non-equivalent** and **faulty**: while the new fault may or may not be identical to the original fault, we expect tests that have failed on P are still more likely to fail on m_f than to pass.

3. **Non-equivalent** and **not faulty**: in which case the fault is fixed by the mutation (with respect to the test suite concerned).

Note that mutating the faulty statement is more likely to cause the tests that failed on P to pass on m_f (case 3) than on m_c because a faulty program is usually fixed by modifying (i.e., mutating) a faulty statement, not a correct one. Therefore, the number of the failing test cases whose results change to pass will be larger for m_f than for m_c.

In contrast, mutating correct statements is not likely to make more test cases pass. Rather, we expect an opposite effect, which is as follows:

Conjecture 2: test cases that used to pass on P are more likely to fail on m_c than on m_f.

Similarly to the case of m_f, the second conjecture is based on an observation that m_c can be either:

1. **Equivalent mutant**, in which case the statement remains correct. Tests that passed with P should still pass with m_c.
2. **Non-equivalent mutant**: by definition, a non-equivalent mutation on a correct statement introduces a fault, which is the original premise of mutation testing.

This second conjecture is based on the observation that a program is more easily broken by modifying (i.e., mutating) a correct statement than by modifying a faulty statement (case 2). Therefore, the number of the passing test cases whose results change to fail will be greater for m_c than m_f.

To summarize, mutating a faulty statement is more likely to cause more tests to pass than the average, whereas mutating a correct statement is more likely to cause more tests to fail than the average (the average case considers both correct and faulty statements). These two conjectures provide the basis for our MUtation-baSEd fault localization technique (MUSE).

4.2 MUSE: Mutation-based Fault Localization

4.2.1 Suspiciousness Metric of MUSE

Based on the two conjectures, we now define the suspiciousness metric for MUSE, μ. For a statement s of P, let $f_P(s)$ be the set of tests that covered s and failled on P, and $p_P(s)$ the set of tests that covered s and passed on P. With respect to a fixed set of mutation operators, let $mut(s) = \{m_1, \dots m_k\}$ be the set of all mutants of P that mutates s with observed changes in test results. After each mutation $m_i \in mut(s)$, let f_{m_i} and p_{m_i} be the set of failing and passing tests on m_i respectively (f_P and p_P defined on P similarly). Given a weight α, the metric μ is defined as follows:

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \right) \tag{4.1}$$

The first term, $\frac{|f_P(s) \cap p_m|}{|f_P|}$, reflects the first conjecture: it is the proportion of tests that failed on P but now pass on a mutant m that mutates s over tests that failed on P. Similarly, the second term, $\frac{|p_P(s) \cap f_m|}{|p_P|}$, reflects the second conjecture, being the proportion of tests that passed on P but now fail on a mutant m that mutates s over tests that passed on P. When averaged over $mut(s)$, they become the probability of test result change per mutant, from failing to passing and vice versa respectively.

Intuitively, the first term correlates to the probability of s being the faulty statement (it increases the suspiciousness of s if mutating s causes failing tests to pass, i.e. increase the size of $f_P(s) \cap p_m$), whereas the second term correlates to the probability of s *not* being the faulty statement (it decreases the suspiciousness of s if mutating s causes passing tests to fail, i.e. increase the size of $p_P(s) \cap f_m$).

Since it is more likely that a passing test case on P will fail on m than a failing test case on P will pass on m (i.e., breaking a program is easier than correcting the program), we expect the average of the second term to be different from that of the first term. In order to balance the two terms, we use the weight α to adjust the average values of the two terms to be the same. Thus, when we subtract

		Coverage of Test Cases (x, y)							Jaccard		Ochiai		Op2	
	int max; **void** setmax(**int** x, **int** y){	TC_1 (3,1)	TC_2 (5,-4)	TC_3 (0,-4)	TC_4 (0,7)	TC_5 (-1,3)	$\|f_P(s)\|$	$\|p_P(s)\|$	Susp.	Rank	Susp.	Rank	Susp.	Rank
s_1:	max = −x; *//should be 'max = x;'*	•	•	•	•	•	2	3	0.40	5	0.63	5	1.25	5
s_2:	**if**(max < y){	•	•	•	•	•	2	3	0.40	5	0.63	5	1.25	5
s_3:	max = y;	•	•		•	•	2	2	0.50	2	0.71	2	1.50	2
s_4:	**if**(x∗y<0)	•	•		•	•	2	2	0.50	2	0.71	2	1.50	2
s_5:	print(''diff.sign'');}		•			•	1	1	0.33	6	0.50	6	0.75	6
s_6:	print(max);}	•	•	•	•	•	2	3	0.40	5	0.63	5	1.25	5
	Test Results	Fail	Fail	Pass	Pass	Pass								

			Test Result Changes							MUSE	
	Statements	Mutants	TC_1 (3,1)	TC_2 (5,-4)	TC_3 (0,-4)	TC_4 (0,7)	TC_5 (-1,3)	$\|f_P(s) \cap p_m\|$	$\|p_P(s) \cap f_m\|$	Suspiciousness	Rank
s_1:	max = −x;	m1: max −= x−1;			P→F			0	1	0.46	1
		m2: max = x;	F→P	F→P				2	0		
s_2:	**if**(max < y){	m3: **if**(!(max<y)){			P→F	P→F	P→F	0	3	0.09	2
		m4: **if**(max==y){	F→P			P→F		1	1		
s_3:	max = y;	m5: max = −y;				P→F	P→F	0	2	-0.16	5
		m6: max = y+1;				P→F	P→F	0	2		
s_4:	**if**(x∗y<0){	m7:**if**(!(x∗y<0))				P→F	P→F	0	2	-0.12	4
		m8:**if**(x/y<0)					P→F	0	1		
s_5:	print(''diff.sign'');}	m9:**return**;					P→F	0	1	-0.08	3
		m10:;					P→F	0	1		
s_6:	print(max);}	m11:printf(0);}				P→F	P→F	0	2	-0.20	6
		m12:;}			P→F	P→F	P→F	0	3		

Figure 4.1: Example of how MUSE localizes a fault compared with different fault localization techniques

the weighted second term from the first term as in Equation 4.1, we get the baseline of value 0. For a faulty statement, the first term is likely to be larger and the second term is likely to be smaller than for a correct statement.

To adjust the average of both terms, the value of α should be calculated as $\frac{f2p}{|mut(P)|\cdot|f_P|} \cdot \frac{|mut(P)|\cdot|p_P|}{p2f}$. Variable $f2p$ and $p2f$ denote the number of test result changes from failure to pass and vice versa between before and after all mutants of P, the set of which is $mut(P)$. Note that α can be calculated without *a priori* knowledge of the faulty statement.

4.2.2 An Working Example

Figure 4.1 presents an example of how MUSE localizes a fault. The PUT is a function called setmax (), which sets a global variable max (initialized to 0) with x if x > y, or with y otherwise. Statement s_1 contains a fault, as it should be max=x. Let us assume that we have five test cases ($tc1$ to $tc5$): the coverage of individual test cases are marked with black bullets (•). TC_1 and TC_2 fail because setmax() updates max with the smaller number, y. The remaining test cases pass. Thus, $|f_P| = 2$ and $|p_P| = 3$.

First, MUSE generates mutants by mutating only one statement at a time. For the sake of simplicity, here we assume that MUSE generates only two mutants per statement, resulting in a total of 12 mutants, $\{m_1, \ldots, m_{12}\}$ (listed under the "Mutants" column of Figure 4.1). Test cases change their results after the mutation as noted in the middle column. For example, TC_1, which used to fail, now passes on the two mutants, $m2$ and $m4$.

Based on the changed results of the test cases, MUSE calculates α as $\frac{f2p}{|mut(P)|\cdot|f_P|} \cdot \frac{|mut(P)|\cdot|p_P|}{p2f} = \frac{3}{12\cdot2} \cdot \frac{12\cdot3}{19} = 0.24$ over 12 mutants ($|mut(P)| = 12$). Since there are three changes from failure to pass, $f2p = 3$ (TC_1 and TC_2 on m_2 and TC_1 on m_4) while $|f_P| = 2$. Similarly, $p2f = 19$ (see the changed results of TC_3, TC_4, and TC_5), while $|p_P| = 3$.

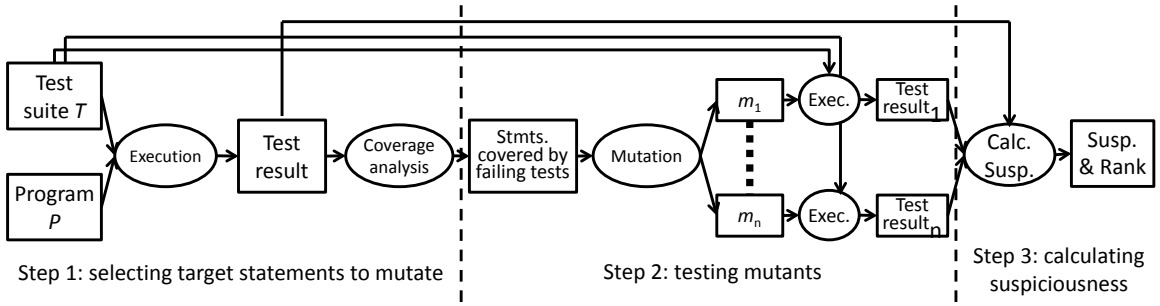

Figure 4.2: Framework of MUtation-baSEd fault localization technique (MUSE)

Using $\alpha = 0.24$, MUSE calculates the suspiciousness of s_1 as $\frac{1}{2}\cdot\{(0/2-0.24\cdot 1/3)+(2/2-0.24\cdot 0/3)\} = 0.46$, where $|f_P(s_1)\cap p_{m_1}| = 0$ and $|p_P(s_1)\cap f_{m_1}| = 1$ for m_1 and $|f_P(s_1)\cap p_{m_2}| = 2$ and $|p_P(s_1)\cap f_{m_2}| = 0$ for m_2. MUSE calculates the suspiciousness scores of the other five statements as 0.09, -0.16, -0.12, -0.08, and -0.20. The suspiciousness of the s_1 is the highest at 0.46, which places it at the top of the ranking. In contrast, Jaccard, Ochiai, and Op2 choose s_3 and s_4 as the most suspicious statements, while assigning the 5th rank to the actual faulty statement s_1. The example shows that MUSE can precisely locate certain faults that the state-of-the-art SBFL techniques cannot.

4.2.3 MUSE Framework

Figure 4.2 shows the framework of MUtation-baSEd fault localization technique (MUSE). There are three major stages: *selection* of statements to mutate, *testing* of the mutants, and *calculation* of the suspiciousness scores.

Step 1: MUSE receives a target program P and a test suite T. After executing T on P, MUSE selects the target statements, i.e. the statements of P that are executed by at least one failing test case in T. We focus on only these statements as those not covered by any failing tests, can be considered not faulty with respect to T.

Step 2: MUSE generates mutant versions of P by mutating each of the statements selected at Step 1. MUSE may generate multiple mutants from a single statement since one statement may contain multiple mutation points [5]. Consequently MUSE tests all generated mutants with T and records the results.

Step 3: MUSE compares the test results of T on P with the test results of T on all mutants. This produces the weight α, based on which MUSE calculates the suspiciousness of the target statements of P[1].

4.3 Empirical Study Setup

We have designed the following three research questions to evaluate the effectiveness of MUSE in terms of the Expense metric [56].

RQ1. Foundation: *To what extent do failing test cases become passing ones on a mutant generated by mutating a faulty statement of a target program, compared with a mutant generated by mutating a correct*

[1]The minimal suspiciousness score is given to the other statements that are not executed by any of the failing test cases.

Table 4.1: Subject programs, their sizes in Lines Of Code (LOC), and the number of failing and passing test cases

Subject program	Faulty Ver.	Fault	Size	$\|f_P\|$	$\|p_P\|$	Description
`flex 2.4.7`	v1	`F_HD_1`	12,423	2	40	Lexical Analyzer
	v7	`F_HD_7`	12,423	1	41	Generator
	v11	`F_AA_3`	12,423	20	22	
`grep 2.2`	v3	`F_DG_4`	12,653	5	175	Pattern
	v11	`F_KP_2`	12,653	177	22	Matcher
`gzip 1.1.2`	v2	`F_KL_2`	6,576	1	211	Compression
	v5	`F_KP_1`	6,576	17	196	Utility
	v13	`F_KP_9`	6,576	3	210	
`sed 1.18`	v1	`F_AG_2`	11,990	42	316	Stream
	v3	`F_AG_17`	11,990	1	357	Editor
	v5	`F_AG_20`	11,990	64	81	
`space`	v19	N/A	9,129	8	145	ADL
	v21	N/A	9,126	1	152	Interpreter
	v28	N/A	9,126	46	107	

statement? Also, to what extent do passing test cases become failing ones on a mutant by mutating a correct statement, compared with a mutant by mutating a faulty statement?

RQ1 is to validate the conjectures in Section 4.1, on which MUSE depends. If these conjectures are valid (i.e., more failing test cases become passing after mutating the faulty statement than a correct one, and more passing test cases become failing after mutating a correct statement than the faulty one), we can expect that MUSE will localize a fault precisely.

RQ2. Precision: *How precise is MUSE, compared with Jaccard, Ochiai, and Op2 in terms of the % of executed statements examined to localize a fault?*

Precision in terms of the % of program statements to be examined is the traditional evaluation criteria for fault localization techniques. RQ2 evaluates MUSE with the Expense metric against the three widely studied SBFL techniques – Jaccard, Ochiai, and Op2. Op2 [47] is *proven* to perform well in Expense metric; Ochiai [50] performs closely to Op2, while Jaccard [30] shows good performance when used with automated progrma repair [55].

To answer the research questions, we performed a series of experiments by applying Jaccard, Ochiai, Op2, and MUSE to the 14 faulty versions in five real-world C programs. The following subsections describe the details of the experiments.

4.3.1 Subject Programs

For the experiments, we used five non-trivial real-world programs including `flex` version 2.4.7, `grep` version 2.2, `gzip` version 1.1.2, `sed` version 1.18, and `space`, all of which are from the SIR benchmark suite [23].

Table 4.1 describes the subject programs including their sizes in Lines of Code, the faulty versions used, and the numbers of failing and passing test cases for each program version/fault pair. From the base versions listed above, we randomly selected three faulty versions from each program except `grep` where a failure is detected only in two faulty versions by the used test suite. `grep v3` and `space v19` have multiple faults and the other versions have one fault per each version. The fault ID of each version is presented in Table 4.1. For `flex`, `grep`, and `space`, we used the coverage-adequate test suite provided by

Table 4.2: The number of target statements, used mutants, and dormant mutants (Those that do not change any test results) per subject

Subject program	Target Stmt.	Used Mutants	Dormant Mutants
`flex v1`	2,769	29,030	7,375
`flex v7`	2,773	28,575	7,411
`flex v11`	2,766	30,366	8,532
`grep v3`	1,982	18,127	10,201
`grep v11`	1,685	12,029	26,425
`gzip v2`	1,448	1,172	835
`gzip v5`	1,419	2,054	1,896
`gzip v13`	1,450	1,238	887
`sed v1`	2,228	13,215	4,813
`sed v3`	2,224	6,307	2,367
`sed v5`	2,151	23,552	0
`space v19`	3,360	14,489	4,919
`space v21`	3,358	9,708	2,790
`space v28`	2,843	13,946	7,443
Average	2318.3	14557.7	6135.3

the SIR benchmark. `flex` and `grep` has only one coverage adequate test suite. For `space`, we randomly chose one coverage adequate test suite out of 1000 coverage-adequate test suites. For `gzip` and `sed`, we use the universe test suite, because the SIR benchmark does not provide a coverage-adequate test suite for the two programs. In addition, we excluded the test cases which caused a subject program version to crash (e.g., segmentation fault), since `gcov` that we used to measure coverage information cannot record coverage information for such test cases.

4.3.2 Experiment Setup

We use `gcov` [1] to measure the statement coverage achieved by a given test case. Based on the coverage information, MUSE generates mutants of the PUT, each of which is obtained by mutating one statement that is covered by at least one failing test case. We use the Proteum mutation tool for the C language [45], which implements the mutation operators defined by Agrawal el al. [5]. For each mutation point in a statement (e.g., a variable or an operator), MUSE generates at most one mutant using Proteum (To generate at most one mutant for each mutation point, we use the option provided by Proteum, which deterministically generates the mutants).

We implemented MUSE, as well as Jaccard, Ochiai, and Op2, in 4,200 lines of C++ code. All experiments were performed on 10 machines equipped with Intel i5 3.6Ghz CPUs and 8GB of memory running Debian Linux 6.05.

4.4 Result of The Experiments

4.4.1 Result of the Mutation

Table 4.2 shows the number of mutants generated per subject program version. On average, MUSE generates 20693.0 (=14557.7+6135.3) mutants per version and uses 14557.7 mutants, while discarding 6135.3 *dormant* mutants, i.e. those for which none of the test cases change their results, on average. [2]

[2] `sed v5` has no dormant mutant because the fault of `sed v5` is non-deterministic one (i.e., it dynamically allocates an smaller amount of memory than necessary through `malloc()`).

Table 4.3: The numbers of the test cases whose results change on the mutants

Subject programs	# of failing tests that pass after mutating:			# of passing tests that fail after mutating:			α
	Correct Stmts. (A)	Faulty Stmts. (B)	(B)/(A)	Correct Stmts. (C)	Faulty Stmts. (D)	(C)/(D)	
`flex v1`	0.0002	1.2727	6155.6	15.7270	8.8182	1.8	0.0009
`flex v7`	0.0002	0.6667	2721.1	16.3644	0.0000	N/A	0.0007
`flex v11`	0.0026	14.2857	5421.3	5.1064	3.5714	1.4	0.0013
`grep v3`	0.1299	0.4792	3.7	30.7825	8.0625	3.8	0.1490
`grep v11`	8.9740	85.8181	9.6	0.1942	0.0000	N/A	5.7939
`gzip v2`	0.0095	0.5625	59.1	113.3410	1.0000	113.3	0.0322
`gzip v5`	0.0611	15.1111	247.2	64.7306	0.1111	582.6	0.0227
`gzip v13`	0.0000	2.7000	N/A	109.2140	0.0000	N/A	0.0141
`sed v1`	0.0095	0.0000	0.0	189.3610	6.1111	31.0	0.0004
`sed v3`	0.0040	0.2500	63.0	238.7950	91.5000	2.6	0.0062
`sed v5`	0.3556	31.8333	89.5	12.6217	12.0690	1.0	0.0365
`space v19`	0.0105	4.6667	444.5	45.7808	13.1667	3.5	0.0057
`space v21`	0.0000	0.3333	N/A	65.6796	1.0000	65.7	0.0002
`space v28`	0.0114	23.0000	2016.5	31.2257	26.5000	1.2	0.0016
Average	0.6835	12.9271	1435.9	67.0660	12.2793	73.4	0.4332

This translates into an average of 6.3 mutants per considered target statement. The mutation and the subsequent testing of all mutants took 10 hours using the 10 machines.

4.4.2 Regarding RQ1: Validity of the Conjectures

Table 4.3 shows the numbers of the test cases whose results change on each mutant of the subject programs. The second and the third columns show the average numbers of failing test cases on P which subsequently pass after mutating a correct statement (i.e. m_c), or a faulty statement (i.e. m_f), respectively. The fifth and the sixth columns show the average numbers of the passing test cases on P which subsequently fail on m_c and m_f respectively. For example, on average, out of the 17 failing test case of `gzip v5`, 0.0611 and 15.1111 failing test cases on `gzip v5` pass on m_c and m_f respectively.

Table 4.3 provides supporting evidence for the conjectures of MUSE discussed in Section 4.1. The number of the failing test cases on P that pass on m_f is 1435.9 times greater than the number on m_c on average, which supports the first conjecture. Similarly, the number of the passing test cases on P that fail on m_c is 73.4 times greater than the number on m_f on average, which supports the second conjecture. Based on these results, we claim that both conjectures are true.

One interesting observation is that the first conjecture seems to be more effective than the second conjecture in its capability to distinguish a faulty statement from correct statements: the average ratio of the number of the failing test cases that change to the passing one on m_f over the number on m_c (i.e. 1435.9) is 19 times greater than the average ratio of the passing test cases that change their results on m_c over the number on m_f (i.e. 73.4).

Table 4.4: Precision of Jaccard, Ochiai, Op2, and MUtation-baSEd fault localization technique (MUSE)

Subject Program	% of executed stmts examined				Rank of a faulty stmt			
	Jaccard	Ochiai	Op2	MUSE	Jaccard	Ochiai	Op2	MUSE
`flex v1`	49.48	45.04	32.01	**0.04**	1,371	1,248	887	**1**
`flex v7`	3.60	3.60	3.60	**0.07**	100	100	100	**2**
`flex v11`	19.76	19.54	13.51	**0.04**	547	541	374	**1**
`grep v3`	1.06	1.01	**0.71**	1.87	21	20	**14**	37
`grep v11`	3.44	3.44	3.44	**1.60**	58	58	58	**27**
`gzip v2`	2.14	2.14	2.14	**0.07**	31	31	31	**1**
`gzip v5`	1.83	1.83	1.83	**0.07**	26	26	26	**1**
`gzip v13`	1.03	1.03	1.03	**0.07**	15	15	15	**1**
`sed v1`	**0.54**	**0.54**	**0.54**	0.90	**12**	**12**	**12**	20
`sed v3`	2.56	2.56	2.56	**0.13**	57	57	57	**3**
`sed v5`	37.84	37.84	37.15	**0.28**	814	814	799	**6**
`space v19`	**0.03**	**0.03**	**0.03**	0.06	**1**	**1**	**1**	2
`space v21`	0.45	0.45	0.45	**0.03**	15	15	15	**1**
`space v28`	11.57	10.66	6.89	**0.04**	329	303	196	**1**
Average	9.67	9.27	7.56	**0.38**	242.64	231.50	184.64	**7.43**

4.4.3 Regarding RQ2: Precision of MUSE in terms of the % of executed statements examined to localize a fault

Table 4.4 presents the precision evaluation of Jaccard, Ochiai, Op2, and MUSE with the proportion of executed statements required to be examined before localizing the fault (i.e. the Expense metric) [3]. The most precise results are marked in bold. Following the ranking produced by MUSE, one can localize a fault after examining 0.38% of the target statements on average. The average precision of MUSE is 25.68 (=9.67/0.38), 24.61 (=9.27/0.38), and 20.09 (=7.56/0.38) times higher than that of Jaccard, Ochiai, and Op2, respectively. In addition, MUSE produces the most precise results for 11 out of the 14 studied faulty versions. This provides quantitative answer to **RQ2**: MUSE can outperform the state-of-the-art SBFL techniques over the Expense metric.

In response to Parnin and Orso [54], we also report the absolute rankings produced by MUSE, i.e. the actual number of statements that need to be inspected before encountering the faulty statement. MUSE ranks the faulty statements of the seven faulty versions (`flex v1,v11`, `gzip v2,v5,v13`, and `space v21,v28`) at the top and ranks the faulty statement of another three versions (`flex v7`, `sed v3`, and `space v19`) among the top three. On average, MUSE ranks the faulty statement among the top 7.43 places, which is 24.86 (=184.64/7.43) times more precise than the best performing SBFL technique, Op2. We believe MUSE is precise enough that its results can be used by a human developer in practice.

4.5 Discussion

Based on the two conjectures we introduced, MUSE not only increases the suspiciousness of potentially faulty statements but also decreases the suspiciousness of potentially correct statements. The results of empirical evaluation show that MUSE can not only significantly outperform the state-of-the-art SBFL techniques, but also provide a practical fault localization solution. MUSE is more than 25 times precise compared to Op2, which is the best known SBFL technique; MUSE also ranks the faulty

[3]The last column of Table 4.3 shows the α values computed for each subject.

statement at the top for seven out of the 14 faulty versions, and among the top three for another three versions. We discuss the superior precision of MUSE in detail in the following sections.

4.5.1 Why does it work well?

As showed in Section 4.4.3, MUSE demonstrates superior precision when compared to the state-of-the-art SBFL techniques. In addition to the finer granularity of statement level, the improvement is also partly because MUSE directly evaluates where (partial) fix can (and cannot) potentially exist instead of predicting the suspiciousness through program spectrum. In a few cases, MUSE actually finds a fix, in a sense that it performs a program mutation that will make all test cases pass (this, in turn, increases the first term in the metric, raising the rank of the location of the mutation). However, in other cases, MUSE finds a *partial* fix, i.e. a mutation that will make only some of previously failing test cases pass. While not as strong as the former case, a partial fix nontheless captures the chain of control and data dependencies that are relevent to the failure and provides a guidance towards the location of the fault.

4.5.2 MUSE and Test Suite Balance

One advantage MUSE has over SBFL is that MUSE is relatively freer from the proportion of passing and failing test cases in a test suite. In contrast, SBFL techniques benefit from having a balanced test suite, and have been augmented by automated test data generation work [25, 32, 59].

MUSE does not require the test suite to have many passing test cases. To illustrate the point, we purposefully calculated MUSE metric without any test cases that passed before mutation (this effectively means that we only use the first term of the metric). On average, MUSE ranked the faulty statement within the top 5.09%, which outperforms SBFL techniques that considered all passing and failing test cases: MUSE is still 1.90 (=9.67/5.09), 1.82 (=9.27/5.09) and 1.49(=7.56/5.09) times more precise than Jaccard, Ochiai, and Op2 respectively.

More interestingly, MUSE does not require the test suite to have many failing test cases. Considering that previous work [32, 59] focused on producing more failing test cases to improve the precision, this is an important observation. We purposefully calculated MUSE metric without any test cases that failed before mutation: although this translates into an unlikely use case scenario, it allows us to measure the differentiating power of the second conjecture in isolation. When only the second term of the MUSE metric is calculated (with $\alpha = 1$), MUSE could still rank the faulty statement among the top 14.62% on average, and among the top 2% for seven out of 14 faulty versions we studied. Intuitively, SBFL techniques require many failing executions to identify where a fault is, whereas MUSE is relatively free from this constraint because it also identifies where a fault *is not*.

This advantage is due to the fact that MUSE utilizes two separate conjectures, each of which is based on the number of failing and passing test cases respectively. Thus, even if a test suite has almost no failing or passing test cases, MUSE can localize a fault precisely.

4.6 Related Work

The idea of generating *diverse program behaviours* to localize a fault more effectively has been utilized by several studies. For example, Cleve and Zeller [21] search for program states that cause the execution to fail by replacing states of a neighbouring passing execution with those of a failing one. If a passing execution with the replaced states no longer passes, relevant statements of the states are reported

as likely faulty statements. Zhang et al. [80], on the other hand, change branch predicate outcomes of a failing execution at runtime to find suspicious branch predicates. A branch predicate is considered suspicious if the changed branch outcome makes a failing execution pass. Similarly, Jeffrey et al. [31] change the value of a variable in a failing execution with the values with other executions; Chandra et al. [19] simulate possible value changes of a variable in a failing execution through symbolic execution. Those techniques are similar to MUSE in a sense that generating diverse program behaviours to localize faults. However, they either *partially* depend on the conjectures of MUSE (some [19, 31, 80] in particular depend on the first conjecture of MUSE) or rely on a different conjecture [21]. Moreover, MUSE does not require any other infrastructure than a mutation tool, because it *directly* changes program source code to utilize the conjectures (Section 4.3.2).

Since mutation operators vary significantly in their nature, mutation-based approaches such as MUSE may not yield itself to theoretical analysis as naturally as the spectrum-based ones, for which hierarchy and equivalence relations have been showed with proofs [74]. In the empirical evaluation, however, MUSE outperformed Op2 SBFL metric [47], which is the known best SBFL technique.

Yoo showed that risk evaluation formulas for SBFL can be automatically evolved using Genetic Programming (GP) [76]. Some of the evolved formulas were proven to be equivalent to the known best metric, Op2 [75]. While current MUSE metrics are manually designed following human intuition, they can be evolved by GP in a similar fashion.

Papadakis and Le-Traon have used mutation analysis for fault localization [53]. However, instead of measuring the impact of mutation on partial correctness as in MUSE (i.e. the conjecture 1), Papadakis and Le-Traon depend on the similarity between mutants in an attempt to detect unknown faults: variations of existing risk evaluation formulas were used to identify suspicious mutants. Zhang et al. [79], on the other hand, use mutation analysis to identify a fault-inducing commit from a series of developer commits to a source code repository: their intuition is that a mutation at the same location as the faulty commit is likely to result in similar behaviours and results in test cases. Although MUSE shares a similar intuition, we do not rely on tests to exhibit similar behaviour: rather, both of MUSE metrics measures what is the *differences* introduced by the mutation. Given the disruptive nature of the program mutation, we believe MUSE is more robust.

Chapter 5. LIL: An Evaluation Metric for Fault Localization Techniques

This chapter presents a new evaluation metric for fault localization techniques, which can measure the aptitude of a localization technique for automated program repair tools as well as human debuggers. The chapter first presents the motivation of the proposed metric by describing the limitation of the exiting evaluation metric that has been used widely for fault localization techniques. The chapter then presents the detailed approach of the proposed metric. The chapter next demonstrates usefulness of the proposed metric through the evaluation of fault localization techniques using the proposed metric, and a case study with an automated program repair tool.

5.1 Motivation

The output of fault localization techniques can be consumed by either human developers or automated program repair techniques. In SBFL literature, the human consumption model assumes the output format of ranking of statements according to their suspiciousness, which is to be linearly followed by humans until identifying the actual faulty statement. Expense metric [56] measures the portion of program statements that need to be inspected until the localization of the fault. It has been widely adopted as an evaluation metric for fault localization techniques [35, 47, 76] as well as a theoretical framework that showed hierarchies between SBFL techniques [74, 75]. However, the Expense metric has been criticised for being unrealistic to be used by a human developer directly [54].

In an attempt to evaluate the precision of SBFL techniques, Qi et al. [55] compared SBFL techniques by measuring the Number of Candidate Patches (NCP) generated by GenProg [68] automated program repair tool, with the given localization information.[1] Automated program repair techniques tend to bypass the ranking and directly use the suspiciousness scores of each statement as the probability of mutating the statement (expecting that mutating a highly suspicious statement is more likely to result in a potential fix) [26, 68]. An interesting empirical observation by Qi et al. [55] is that Jaccard [30] produced lower NCP than Op2 [47], despite having been proven to always produce a lower ranking for the faulty statement than Op2 [74]. This is due to the actual distribution of the suspiciousness score: while Op2 produced higher ranking for the faulty statement than Jaccard, it assigned almost equally high suspiciousness scores to some correct statements. On the other hand, Jaccard assigned much lower suspiciousness scores to correct statements, despite ranking the faulty statement slightly lower than Op2. This illustrates that evaluation and theoretical analysis based on the linear ranking model is not applicable to automated program repair techniques.

[1] Essentially this measures the number of fitness evaluation for the Genetic Programming part of GenProg; hence the lower the NCP score is, the more efficient GenProg becomes, and in turn the more effective the given localization technique is.

5.2 Locality Information Loss (LIL) Metric

We propose a new evaluation metric, called LIL, that does not suffer from this discrepancy between the two consumption models (human and automated program repair tool). LIL metric can measure the aptitude of fault localization techniques for automated program repair techniques as it measures the effectiveness of localization in terms of information loss rather than the behavioural cost of inspecting a ranking of statements. LIL metric essentially captures the essence of the entropy-based formulation of fault localization [77] in the form of an evaluation metric.

Let S be the set of n statements of the Program Under Test, $\{s_1, \ldots, s_n\}$, s_f, $(1 \leq f \leq n)$ being the single faulty statement. Without losing generality, we assume that output of any fault localization technique τ can be normalized to [0, 1]. Now suppose that there exists an ideal fault localization technique, $\mathcal{L}$, that can always pinpoint s_f as follows:

$$\mathcal{L}(s_i) = \begin{cases} 1 & (s_i = s_f) \\ \epsilon & (0 < \epsilon \ll 1, s_i \in S, s_i \neq s_f) \end{cases} \tag{5.1}$$

Note that we can convert outputs of fault localization techniques that do not use suspiciousness scores in a similar way: if a technique τ simply reports a set C of m statements as candidate faulty statements, we can set $\tau(s_i) = \frac{1}{m}$ when $s_i \in C$ and $\tau(s_i) = \epsilon$ when $s_i \in S - C$.

We now cast the fault localization problem in a probabilistic framework as in the previous work [77]. Since the *suspiciousness* score of a statement is supposed to correlate to the likelihood of the statement containing the fault, we convert the suspiciousness score given by a fault localization technique, $\tau : S \rightarrow [0, 1]$, into the probability of any member of S containing the fault, $P_\tau(s)$, as follows:

$$P_\tau(s_i) = \frac{\tau(s_i)}{\sum_{i=1}^{n} \tau(s_i)}, (1 \leq i \leq n) \tag{5.2}$$

This converts suspiciousness scores given by any τ (including $\mathcal{L}$) into a probability distribution, P_τ. The metric we propose is the Kullback-Leibler divergence [37] of P_τ from $P_\mathcal{L}$, denoted as $D_{KL}(P_\mathcal{L}||P_\tau)$: it measures the information loss that happens when using P_τ instead of $P_\mathcal{L}$ and is calculated as follows:

$$D_{KL}(P_\mathcal{L}||P_\tau) = \sum_i \ln \frac{P_\mathcal{L}(s_i)}{P_\tau(s_i)} P_\mathcal{L}(s_i) \tag{5.3}$$

We call this as Locality Information Loss (LIL). Kullbacl-Leibler divergence between two given probability distribution P and Q requires the following: both P and Q should sum to 1, and $Q(s_i) = 0$ implies $P(s_i) = 0$. We satisfy the former by the normalization in Equation 5.2 and the latter by always substituting 0 with ϵ *after* normalizing τ[2] (because we cannot guarantee the implication in our application). When these properties are satisfied, $D_{KL}(P_\mathcal{L}||P_\tau)$ becomes 0 when $P_\mathcal{L}$ and P_τ are identical. As with the Expense metric, the lower the LIL value is the more accurate the fault localization technique is. Based on Information Theory, LIL has the following strengths compared to the Expense metric:

- **Expressiveness**: unlike the Expense metric that only concerns the actual faulty statement, LIL also reflects how well the suspiciousness of non-faulty statements have been supressed by a fault localization technique. That is, LIL can be used to explain the results of Qi et al. [55] quantitatively.

[2] ϵ should be smaller than the smallest normalized non-zero suspiciousness score by τ.

Table 5.1: Expense and LIL of Jaccard, Ochiai, Op2, and MUtation-baSEd fault localization technique (MUSE)

Subject Program	% of executed stmts examined				Locality Information Loss (LIL)			
	Jaccard	Ochiai	Op2	MUSE	Jaccard	Ochiai	Op2	MUSE
`flex v1`	49.48	45.04	32.01	**0.04**	8.25	7.79	7.64	**1.28**
`flex v7`	3.60	3.60	3.60	**0.07**	5.65	6.43	7.49	**1.22**
`flex v11`	19.76	19.54	13.51	**0.04**	7.27	7.38	7.29	**1.59**
`grep v3`	1.06	1.01	**0.71**	1.87	**5.15**	5.57	6.19	5.92
`grep v11`	3.44	3.44	3.44	**1.60**	**5.25**	6.06	5.30	7.19
`gzip v2`	2.14	2.14	2.14	**0.07**	5.10	4.45	6.23	**1.66**
`gzip v5`	1.83	1.83	1.83	**0.07**	4.22	4.51	5.12	**1.88**
`gzip v13`	1.03	1.03	1.03	**0.07**	2.99	3.48	5.68	**0.70**
`sed v1`	**0.54**	**0.54**	**0.54**	0.90	**4.08**	4.83	5.63	6.72
`sed v3`	2.56	2.56	2.56	**0.13**	6.66	6.37	6.96	**2.66**
`sed v5`	37.84	37.84	37.15	**0.28**	7.10	7.19	7.11	**4.80**
`space v19`	**0.03**	**0.03**	**0.03**	0.06	5.12	5.76	6.52	**2.15**
`space v21`	0.45	0.45	0.45	**0.03**	4.80	5.79	7.45	**0.40**
`space v28`	11.57	10.66	6.89	**0.04**	7.14	7.21	7.05	**1.96**
Average	9.67	9.27	7.56	**0.38**	5.63	5.91	6.55	**2.87**

- **Flexibility**: unlike the Expense metric that only concerns a single faulty statement, LIL can handle multiple locations of faults. For m faults (or for a fault that consists of m different locations), the distribution $P_{\mathcal{L}}$ will simply show not one but m spikes, each with $\frac{1}{m}$ as height.
- **Applicability**: Expense metric is tied to fault localization techniques that produce rankings, whereas LIL can be applied to any fault localization technique. If a technique assigns suspiciousness scores to statements, it can be converted into P_τ; if a technique simply presents one or more statements as candidate fault location, P_τ can be formulated to have corresponding peaks.

5.3 Evaluation using LIL metric

We evaluate the effectiveness of SBFL techniques, Jaccard, Ochiai, Op2, and the proposed technique MUSE in terms of LIL on the same subject programs used in Section 4.3. We report the results of traditional metric (i.e., expense metric) as well as the results of LIL metric, in order to compare the usefulness of both two metrics. We then perform a case study to show that the LIL metric is better at predicting the performance of a fault localization technique for automated program repair tools than the traditional ranking model. Through the evaluation of fault localization techniques and the case study, we will demonstrate not only the advantage of LIL metric, but also the superior precision of MUSE (Chapter 4) in terms of LIL metric.

5.3.1 Evaluation of Fault Localization Techniques

The LIL column of Table 5.1 shows the precision of Jaccard, Ochiai, Op2, and MUSE in terms of the LIL metric, computed with $\epsilon = 10^{-16}$. The best results (i.e. the lowest values) are marked in bold. The LIL metric value of MUSE is 2.87 on average, which is 1.96 (=5.63/2.87), 2.06 (=5.91/2.87), and 2.28 (=6.55/2.87) times more precise than those of Jaccard, Ochiai, and Op2. In addition, the LIL metric

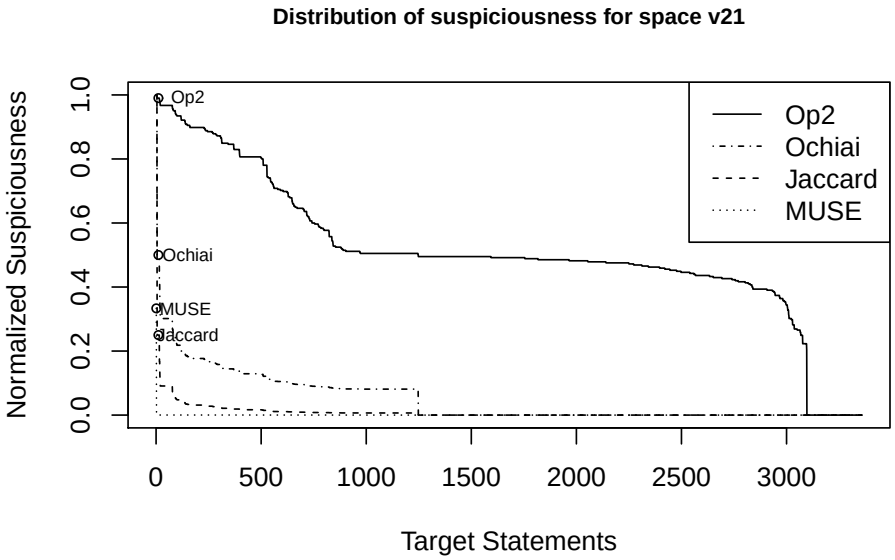

Figure 5.1: Normalized suspiciousness scores from `space v21` in descending order

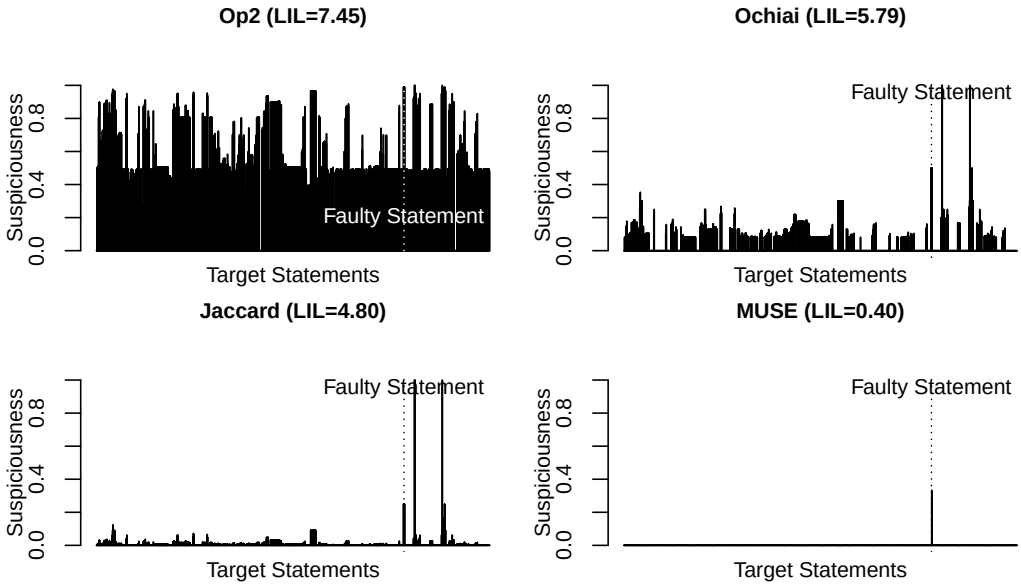

Figure 5.2: Comparison of distributions of normalized suspiciousness score across target statements of `space v21`

values of MUSE are the smallest ones on the eleven out of the 14 subject program versions. MUSE outperforms the state-of-the-art SBFL techniques over the newly proposed LIL metric.

One interesting observation is that MUSE produces Expense and LIL values that correlates relatively well. The versions whose absolute ranking of faulty statement is equal to or less than 3, and whose LIL metric is less than 2.66, are the following 10 versions: `flex v1,v7,v11`, `gzip v2,v5,v13`, `sed v3`, and `space v19,v21,v28`. For another three versions (`grep v3,v11` and `sed v1`), both the Expense and LIL metric values perform worse than the other techniques, although not significantly.

In contrast, Expense and LIL metric often do not agree with each other for the SBFL techniques. Consider `space v21`: Jaccard, Ochiai, and Op2 produces the same Expense value of 0.45%. However,

Table 5.2: Expense, LIL, and NCP scores on `look utx 4.3`

Fault Localization Technique	% of executed stmts examined	Locality Information Loss (LIL)	Average of NCP over 100 runs
MUSE	11.25	3.52	25.3
Op2	42.50	3.77	31.0
Ochiai	42.50	3.83	32.2
Jaccard	42.50	3.89	35.5

their LIL values are all different (Jaccard: 4.80 < Ochiai: 5.79 < Op2: 7.45). A similar pattern is observed in other subject versions (`flex v7`, `grep v11`, `gzip v2,v5,v13`, `sed v1,v3`, `space v19,v21`). In case of `grep v3`, the Expense metric and the LIL metric directly conflict with each other. With respect to the Expense values, Op2 produces the best result, while Jaccard produces the worst result (Jaccard: 1.06 > Ochiai: 1.01 > Op2: 0.71). However, with respect to the LIL values, Jaccard produces the best result, while Op2 the worst (Jaccard: 5.15 < Ochiai: 5.57 < Op2: 6.19).

Figure 5.1 illustrates this phenomenon in more detail. It plots the normalized suspiciousness scores for each target statement of `space v21` in a descending order [3]. The circles indicate the location of the faulty statement. While all techniques assign, to the faulty statement, suspiciousness values that rank near the top, it is the suspiciousness of correct statements that differentiates the techniques. When normalized into $[0, 1]$, MUSE assigns values less than 0.00024 to all correct statements. In contrast, the SBFL techniques assign values much higher than 0. For example, 4.8% of the target statements are assigned suspiciousness higher than 0.9 by Op2, while 37.2% are assigned values higher than 0.5. Figure 5.2 presents the distribution of suspiciousness in `space v21` for individual techniques to make it easier to observe the differences. It intuitively illustrates the strength of the LIL metric over the Expense metric.

This independently confirms the results obtained by Qi et al. [55]. Our new evaluation metric, LIL, confirms the same observation as Qi et al. by assigning Jaccard a lower LIL value of 4.80 than that of Op2, 7.45 (see Section 5.1 and 5.2 for more details).

5.3.2 A Case Study: LIL metric and Automated Bug Repair

We performed a small case study with the GenProg-FL tool by Qi et al., which is a modification of the original GenProg tool. We applied Jaccard, Ochiai, Op2, and MUSE, to GenProg-FL in order to fix `look utx 4.3`, which is one of the subject programs recently used by Le Goues et al. [27]. GenProg-FL [55] measures the NCP (Number of Candidate Patches generated before a valid patch is found in the repair process) of each fault localization technique where the suspiciousness score of a statement s is used as the probability to mutate s.

Table 5.2 shows the Expense (i.e., % of executed stmts examined), the LIL and the NCP scores on `look utx 4.3` by the four fault localization techniques we have evaluated. For the case study, we generated 30 failing and 150 passing test cases randomly and used the same experiment parameters as in GenProg-FL [55] (we obtained the average NCP score from 100 runs). Table 5.2 demonstrates that the LIL metric is useful to evaluate the effectiveness of a fault localization technique for the automatic repair of `look utx 4.3` by GenProg-FL: the LIL scores (MUSE : 3.52 < Op2 : 3.77 < Ochiai : 3.83 <

[3] The normalized suspiciousness of a statement s in a fault localization technique τ, $norm\_susp_\tau(s)$ is computed as $(susp_\tau(s) - min(\tau))/(max(\tau) - min(\tau))$ where $min(\tau)$ and $max(\tau)$ is the minimum and maximum observed suspiciousness for all statements [55].

Jaccard : 3.89) and the NCP scores (MUSE : 25.3 < Op2 : 31.0 < Ochiai : 32.2 < Jaccard : 35.5) are in agreement.

A small LIL score of a localization technique indicates that the technique can be used to perform more efficient automated program repair. In contrast, the Expense metric values did not provide any information for the three SBFL techniques. We plan to perform further empirical study to support the claim.

Chapter 6. Conclusion and Future Work

Fault localization is the most expensive phase in the whole debugging activity. To reduce the cost for fault localization, thus, we have proposed techniques that assist a developer to effectively locate faults in target program. The proposed techniques automatically rank statements in target program according to their suspiciousness, which is computed by utilizing program transformation, mutation analysis, and dynamic information gathered from test executions. The ranked statements can be reported to a developer to reduce her manual efforts to locate faults. For example, a developer can find a faulty statement in the subject programs we used, by only examining 7.4 statements on average while following the order of statements in the ranking produced by the proposed technique MUSE. Therefore, we believe that the proposed techniques can significantly reduce the cost for software fault localization.

6.1 Summary

In this dissertation, we first investigated the fundamentals of previous fault localization techniques and their limitations. In these investigations, we described that the promising fault localization approach, called SBFL, suffers from the negative effect Coincidentally Correct Test Cases (CCTs) (i.e., test cases that pass despite executing faults), and the coarse granularity of block level of SBFL. Second, we developed a new fault localization technique, called FIESTA, which mitigates the negative effect of CCTs to improve the precision of SBFL. It mitigates the CCT problem by using the newly proposed techniques - *fault weight metric* and *debuggability transformation*. We evaluated FIESTA on 12 programs including 5 real-world programs (consist of 173 to 12,653 LOC), and showed that FIESTA is relatively more precise than SBFL techniques upto 49%, on average. Third, we developed another new fault localization technique, called MUSE, which utilizes mutation analysis to overcome the limitations of SBFL. MUSE localizes faults very precisely based on the two conjectures, which essentially capture the different characteristics of two groups of mutants - mutants on faulty statements and mutants on correct statements. Empirical evaluation of MUSE on 5 real-world programs (consist of 6,576 to 12,653 LOC) showed that MUSE ranks a fault among the top 7.4 places on average, which is about 25 times more precise than the state-of-art SBFL technique, Op2. Fourth, we proposed a new evaluation metric, called Locality Information Loss (LIL), for fault localization techniques. LIL actually measures the loss of information between the true locality of fault and the predicted locality from a localization technique, using information theory. Evaluation of fault localization techniques using LIL metric and a case study showed that LIL metric can be better at predicting the performance of a fault localization technique for automated program repair tool.

6.2 Future Work

Our future work will focus on developing techniques that improve the effectiveness and the efficiency of the proposed technique MUSE, which showed more promising results than FIESTA. In addition, we plan to apply MUSE on different domains.

6.2.1 Improving the Effectiveness

Although MUSE localizes faults very precisely on our subject programs, the technique should be developed further to localize particular faulty statements the current MUSE cannot localize. Since MUSE localizes faults by observing differences between test results of target program and those of mutants, faulty statements where no mutants are generated cannot be located by MUSE. One possible way to solve the problem is to define additional mutation operators to mutate statements that cannot be mutated by current mutation operators. Another possible way is to combine MUSE with other fault localization techniques. For example, MUSE can be easily combined with SBFL techniques that utilize program spectra. If no mutants are generated for certain statements, the suspiciousness of the statements can be computed by utilizing SBFL formulas [30, 36, 47, 50]. These approaches will enable MUSE to localize faulty statements where no mutants are generated.

6.2.2 Improving the Efficiency

The proposed technique MUSE utilizes mutation analysis, which can consume much time due to the large number of mutants and test cases used. Thus, appropriate optimization techniques that reduce the time complexity should be developed. We plan to develop optimization techniques in the following four ways.

- Mutant sampling which can reduce the number of mutants used for fault localization. Similar to the mutation analysis for software testing [52], we can sample certain number of mutants per each statement or per each mutation operator, to reduce the number of mutants used. If MUSE with a certain ratio of whole mutants shows comparable performance to MUSE with whole mutants, mutant sampling can enable to improve the efficiency without losing the effectiveness.

- Defining sufficient mutation operators to localize faults effectively. The current implementation of MUSE uses all mutation operators defined in [45], thus it can generate a lot of mutants. However, some of the mutation operators used in MUSE may not be necessary for effective fault localization. Similar to the intuitions of the studies that try to decrease the cost of mutation analysis for software testing [14, 51, 62], we can define sufficient mutation operators for effective fault localization of MUSE through the robust empirical study.

- Generating or selecting certain test cases that are sufficient to localize faults effectively. As the required time for mutation analysis increases proportional to the the number of test cases used, reducing the number of test cases can improve the efficiency. Thus, techniques that generate or select certain test cases for effective fault localization of MUSE could be developed.

- Combining MUSE with program slicing techniques to reduce the number of mutants. We can reduce the number of mutants used for MUSE by reducing the number of target statements that are mutated through program slicing. For instance, if certain statements are not contained in a dynamic slice, which is computed by a dynamic slicing technique [81] according to the given slicing criterion (i.e., a value of variable that manifests failures), the certain statements will not contain faults. Thus, the number of generated mutants will be reduced since those statements not need to be mutated.

6.2.3 Applications on Different Domains

As MUSE requires source code of target program, mutation operators, and a test suite, MUSE can be extended to localize faults in other types of applications such as web applications and concurrent applications. If target source code, appropriate mutation operators mutating target statements, and a test suite are provided, we believe that MUSE can precisely localize faults in any types of applications as it does in sequential applications written in C programming language.

References

[1] GCOV. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[2] GDB: The GNU project debugger. `https://www.gnu.org/software/gdb/`.

[3] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund. Refining spectrum-based fault localization rankings. In *ACM Symposium on Applied Computing*, 2009.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing*, 2006.

[5] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the c programming language. Technical report, Purdue University, 1989. Technical Report SERC-TR-120-P.

[6] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[7] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *IEEE Software Reliability Engineering*, 1995.

[8] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation*, 1990.

[9] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *International Symposium on Software Testing and Analysis*, 2010.

[10] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *International Conference on Software Engineering*, 2010.

[11] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *International Symposium on Software Testing and Analysis*, 2010.

[12] G. K. Baah, A. Podgurski, and M. J. Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *European Software Engineering Conference/Foundations of Software Engineering*, 2011.

[13] A. Bandyopadhyay. Improving spectrum-based fault localization using proximity-based weighting of test cases. In *Automated Software Engineering*, 2011.

[14] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[15] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *International Conference on Software Engineering*, 2006.

[16] Á. Beszedes, T. Gergely, Z. M. Szabo, J. Csirik, and T. Gyimothy. Dynamic slicing method for maintenance of large c programs. In *Europen Conference on Software Maintenance and Reengineering*, 2001.

[17] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, 1980.

[18] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Summer School on Computer Program Testing*, 1981.

[19] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In *International Conference on Software Engineering*, 2011.

[20] R. Chillarege, W. Kao, and R. G. Condit. Defect type and its impact on the growth curve. In *International Conference on Software Engineering*, 1991.

[21] H. Cleve and A. Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, 2005.

[22] N. Digiuseppe and J. A. Jones. On the influence of multiple faults on coverage-based fault localization. In *International Symposium on Software Testing and Analysis*, 2011.

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

[24] J. A. Duraes 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.

[25] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving effectiveness of spectra-based fault localization using specifications. In *Automated Software Engineering*, 2012.

[26] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, 2012.

[27] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *ACM Transaction on Software Engineering and Methodology*, 38(1):54–72, 2012.

[28] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *ACM SIGPLAN Notices*, 1998.

[29] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *International Conference on Software Engineering*, 1994.

[30] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[31] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *International Symposium on Software Testing and Analysis*, 2008.

[32] W. Jin and A. Orso. F3: fault localization for field failures. In *International Symposium on Software Testing and Analysis*, 2013.

[33] J. A. Jones. *Semi-automatic fault localization.* PhD thesis, Georgia Institute of Technology, 2008.

[34] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *International Symposium on Software Testing and Analysis*, 2007.

[35] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Automated Software Engineering*, 2005.

[36] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, 2002.

[37] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[38] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Program Language Design and Implementation*, 2003.

[39] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Program Language Design and Implementation*, 2005.

[40] M. A. Linton. The evolution of dbx. In *USENIX Summer*, 1990.

[41] C. Liu and J. Han. Failure proximity: a fault localization-based approach. In *International Sympo-*

sium on Foundations of Software Engineering, 2006.

[42] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midki. Sober: statistical model-based bug localization. In *Foundations of Software Engineering*, 2005.

[43] C. Liu, X. Zhang, and J. Han. A systematic study of failure proximity. *IEEE Transactions on Software Engineering*, 34(6):826–843, November 2008.

[44] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[45] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. da Silva Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero. Mutation testing for the new century. chapter Proteum: a family of tools to support specification and program testing based on mutation, pages 113–116. Kluwer Academic Publishers, 2001.

[46] W. Masri and R. A. Assi. Cleansing test suites from coincidental correctness to enhance fault-localization. In *International Conference on Software Testing, Verification and Validation*, 2010.

[47] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering Methodology*, 20(3):11:1–11:32, August 2011.

[48] National Institute of Standards and Technology (NIST), 2002. Software Errors Cost U.S. Economy $59.5 Billion Annually.

[49] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *CC*, 2002.

[50] A. Ochiai. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bulletin of the Japanese Society of Scientific Fisheries*, 22(9):526–530, 1957.

[51] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[52] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. 2001.

[53] M. Papadakis and Y. L. Traon. Using mutants to locate "unknown" faults. In *International Workshop on Mutation Analysis*, 2012.

[54] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, 2011.

[55] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

[56] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering*, 2003.

[57] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *European Software Engineering Conference*, 1997.

[58] D. J. Richardson and M. C. Thompson. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, 1993.

[59] J. Rößler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *International Symposium on Software Testing and Analysis*, 2012.

[60] J. H. Saltzer and M. F. Kaashoek. *Topics in the engineering of computer systems*. MIT, Department of Electrical engineering and computer science, 1980.

[61] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *International Conference on Software Engineering*, 2009.

[62] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*, 2008.

[63] Standard Coordinating Commitee IEEE. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[64] H. Steinhaus. Sur la division des corp materiels en parties. *Bull. Acad. Polon. Sci*, 1:801–804, 1956.

[65] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[66] I. Vessey. Expertise in debugging compute programs. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[67] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *International Conference on Software Engineering*, 2009.

[68] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, 2009.

[69] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[70] M. Weiser. Programmers use slices when debugging. *CACM*, 25(7):446–452, 1982.

[71] W. E. Wong and V. Debroy. A survey of software fault localization. Technical report, University of Texas at Dallas, 2009. Technical Report UTDCS-45-09.

[72] W. E. Wong, V. Debroy, and B. Choi. A family code coverage-based heuristics for effective fault localization. *Journal of Software Systems*, 83(2):188 –208, sept. 2010.

[73] W. E. Wong and Y. Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79(2):891–903, 2006.

[74] X. Xie, T. Y. Chen, F. C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transaction on Software Engineering and Methodology*, 2013(to appear).

[75] X. Xie, F.-C. Kuo, T. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *International Symposium on Search-Based Software Engineering*, 2013.

[76] S. Yoo. Evolving human competitive spectra-based fault localisation techniques. In *International Symposium on Search-Based Software Engineering*, 2012.

[77] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering Methodology*, 22(3):19:1–19:29, 2013.

[78] A. Zeller. Isolating cause-effect chains from computer programs. In *European Software Engineering Conference/Foundations of Software Engineering*, 2002.

[79] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2013.

[80] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *International Conference on Software Engineering*, 2006.

[81] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *International Conference and Software Engineering*, 2003.

[82] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced

ordered binary decision diagrams. In *International Conference on Software Engineering*, 2004.

[83] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *International Conference on Machine Learning*, 2006.

Summary

Effective Software Fault Localization using Dynamic Program Behaviors

프로그램 오류의 원인을 찾는 것은 소프트웨어 디버깅 과정 중 가장 많은 비용을 요구 하는 과정 중 하나로 알려져 있다. *결함 위치추정*이라고 불리는 이 과정은 개발자가 직접 테스트 케이스의 실행 정보 등을 이용하여 결함의 위치를 추정해야 하기 때문에 가장 많은 시간과 노력을 요구한다. 따라서, 본 논문에서는 결함 위치추정에 소모되는 비용을 줄이기 위한 두 가지 자동 결함 위치추정 기법을 제안하였다. 제안된 결함 위치추정 기법들은 대상 프로그램의 각 구문들이 결함일 가능성을 계산하여 그 가능성에 따라 각 구문들을 순위화 하는데, 이 때 각 구문이 결함일 가능성은 대상 프로그램, 변형된 대상 프로그램, 변환된 대상 프로그램의 동적 정보를 활용하여 계산된다. 개발자는 순위화된 구문을 순서대로 검사함으로써 결함 위치추정에 드는 비용을 줄일 수 있다.

본 논문에서는, 먼저 이전 결함 위치추정 기법들의 근본적 접근방법과 한계에 대해서 조사하였다. 이 조사에서, 전도유망한 연구 방향 중 하나인 스펙트럼에 기반한 결함 위치추정 기법(Spectrum-based Fault Localization (SBFL))의 한계에 대해서 설명했다. 그 한계는, 우연히 성공한 테스트 케이스 (Coincidentally Correct Test Cases (CCTs))로 인한 부정적 효과와 SBFL 기법의 정제되지 않은 블락 단위 수준이었다. 두 번째로, 본 논문에서는 CCT의 부정적 효과를 감소시키는 결함 위치추정 기법인 FIESTA를 제안하였다. FIESTA 기법은 각 테스트 케이스가 결함을 실행했을 확률을 나타내는 'Fault weight'척도와 CCT의 수를 줄이는 'Debuggability transformation'기법을 이용하여 결함과 실패 실행간의 상관관계를 향상 시킨다. 결함과 실패 실행간의 향상된 상관관계는, FIESTA가 기존의 SBFL 기법에 비해 더 정확하게 결함의 위치를 추정하도록 만든다. 12개의 C 프로그램(173 ∼ 12,653 LOC 로 구성)에 대한 실험을 통해 FIESTA가 대표적 SBFL 기법인 Tarantula에 비해 상대적으로 49% 더 정확하게 결함의 위치를 추정하는 것을 확인할 수 있었다. 세 번째로, 우리는 변형 분석을 활용하여 결함의 위치를 추정하는 새로운 종류의 결함 위치추정 기법인 MUSE를 제시했다. MUSE 기법은 본 논문에서 제시한 두 개의 가정에 기반하여 결함의 위치를 추정한다. 두 개의 가정은 기본적으로 결함 구문을 변형시킨 변형 프로그램들과 결함 구문이 아닌 구문을 변형시킨 변형 프로그램들을 실행한 테스트 케이스의 결과가 서로 다른 특성을 보임을 이용한다. 5개의 실제 C 프로그램들(6,576 ∼ 12,653 LOC 로 구성)에 대한 MUSE의 성능평가 결과로부터 MUSE는 결함 구문의 순위를 평균적으로 상위 7.4등 이내로 매기는것을 확인할 수 있었다. 이 실험결과는 지금까지 알려진 가장 정확한 SBFL 기법인 Op2 보다 평균적으로 약 25배 더 정확한 결과다. 네 번째로, 결함 위치추정 기법의 성능을 평가하는 새로운 평가 척도인 LIL 척도를 제안하였다. LIL 은 정보 이론에 기반하여 이상적인 위치추정 결과와 어떤 결함 위치추정 기법에 의해 주어진 위치추정 결과 사이의 정보 손실을 측정한다. LIL을 이용한 다수의 결함 위치추정 기법에 대한 평가와 자동 프로그램 수리 도구를 이용한 사례 연구를 통해, LIL이 결함 위치추정 기법의 자동 프로그램 수리 도구에 대한 성능을 측정하는데 유용할 수 있다는 것을 확인하였다.

감 사 의 글

석사 연구기간동안 위로가 되고 힘이 되어준 아버지, 어머니, 형, 그리고 할머니, 할아버지께 감사드립니다. 처음하는 타지 생활과 대학교와는 다른 대학원 생활에 낯설어 하는 저를 위해 항상 기도하고 염려했던 가족이 있었기에, 스스로를 격려하고 다독이면서 석사 연구과정을 무사히 끝내고 이 논문을 완성할 수 있었습니다.

연구의 큰 방향을 잡아주시고, 연구를 위한 논리적인 사고를 가지도록 저를 변화시키고, 연구를 옳은 방향으로 이끌도록 걱정하고 노력하셨던 김문주 교수님께 감사드립니다. 소프트웨어 테스팅과 결함 위치 추정(fault localization)에 대한 명확한 개념이 없던 저를 훈련시키고, 새로운 관점에서 문제를 바라볼 수 있도록 지도하셨습니다. 특히, 비록 연구의 결과가 좋지 않더라도 항상 긍정적인 마음으로 계속 도전하고자 했던 교수님의 자세는 연구뿐 아니라, 제가 앞으로 살아갈 삶에도 긍정적인 영향을 미쳤습니다. 그리고 ICST 학회에 MUSE기법을 논문으로 제출하기 위해 같이 연구했던 UCL의 유신 교수님께도 감사합니다. LIL metric이라는 새로운 아이디어를 제안하고, 저의 아이디어를 좀 더 명확하게 표현하도록 이끌어 주신 덕분에, ICST에 논문을 제출 할 수 있었습니다.

언제나 진취적이고 창의적인 논의와 사고로 저를 격려하고, MUSE 기법을 개발할 수 있도록 조언을 아끼지 않은 홍신 박사과정에게 감사합니다. 연구 분야가 서로 다름에도 불구하고, 마치 자신의 분야처럼 열정을 가지고 함께 고민했던 수많은 순간들이 없었다면, MUSE 기법을 개발할 수 없었을 것입니다. 그리고, 함께 결함 위치추정 기법을 연구한 김윤호 박사과정에게도 감사합니다. 제 머릿속의 아이디어를 다른사람에게 명확하게 전달할 능력이 없었던 석사 1년차에, 제 아이디어를 끄집어내서 명확하게 만들어준 김윤호 학생이 없었다면, FIESTA 기법을 석사 1년차가 채 다 끝나가지 않을 무렵에 빠르게 개발할 수 없었을 것입니다. 또한, 석사 논문을 좀 더 짜임새가 있도록 도와주신 문영주 박사님, 바쁜 와중에도 저의 석사 논문 완성을 위해 자료를 제공해준 김영주 졸업생, 석사 연구기간 동안 서로 위로와 힘이 되었던 안재민, 박용배, 연광흠, 김대철 학생에게도 진심으로 감사합니다.

같은 석사과정 학생으로, 함께 연구하는 분야에 대해 고민하고 앞으로 나아갈 방향에 대해서 얘기하면서 저의 시야를 넓혀준 많은 친구들에게 감사합니다. 친구들을 통해서 연구분야의 다양성, 전혀 다른 관점으로 문제를 바라볼 수 있는 관점을 배울 수 있었고, 전산학 전반에 대해 더욱 깊은 관심과 흥미를 가질 수 있었습니다. 그리고, 대전에서 서울 혹은 대구로 갈 때 마다 항상 반겨주고, 서로의 생각과 고민을 나눌 수 있었던 친구와 형들에게도 감사합니다. 그분들의 대화와 격려, 그리고 도움이 없었다면, 석사연구를 성공적으로 끝낼 수 없었을 겁니다.

마지막으로, 제가 가야할 길을 벗어나도 저를 항상 옳은 길로 이끄시고, 저와 언제나 함께하시는 하나님께 감사합니다. 저의 기도뿐 아니라, 부모님, 할머니, 할아버지의 기도에 응답해주신 하나님이 계셨기에 지금의 제가 이 자리에 있을 수 있습니다.

제가 했던 2년간의 석사 연구가, 우리가 사는 이 세상에 진정으로 도움이 되기를 진심으로 기원합니다.

이 력 서

이 름 : 문 석 현

생 년 월 일 : 1987년 3월 20일

출 생 지 : 대구광역시 동구 신암동

본 적 지 : 대구광역시 수성구 범어2동 396-5번지 동남빌라 101호

주 소 : 대구광역시 수성구 범어2동 396-5번지 동남빌라 101호

E-mail 주 소 : seokhyeon.mun@gmail.com

학 력

2003. 3. – 2006. 2. 대구경신고등학교

2006. 3. – 2012. 2. 경북대학교 IT대학 컴퓨터학부 (B.S.)

2012. 3. – 2014. 2. 한국과학기술원 전산학과 (M.S.)

경 력 및 수 상 이 력

2012. 6. – 2012. 8. KOG Games 인턴

2013. 6. 27 제 32회 한국정보과학회 학생논문경진대회 최우수상

2013. 12. 19 한국과학기술원 전산학과 석사 워크샵 우수 논문상

학 회 활 동

1. **S.Moon**, Y.Kim, M. Kim, S. Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization, IEEE International Conference on Software Testing, Verification, and Validation (ICST 2014), to appear (**acceptance rate: 28%**).
2. **S.Moon**, Y.Kim, M. Kim, FEAST: An Enhanced Fault Localization Technique using Probability of Test Cases Executing Faults, Journal of KIISE: Software and Applications, Vol 40, Num 10, Oct 2013 (**invited paper**).
3. **S.Moon**, Y.Kim, M. Kim, An Enhanced Fault Localization Technique using Probability of Test Cases Executing Faults, Korea Conference on Software Engineering (KCSE), Jan 30 – Feb 1, 2013.

연 구 업 적

1. **S.Moon**, Y.Kim, M. Kim, FIESTA: Effective Fault Localization to Mitigate the Negative Effect of Coincidentally Correct Tests, in preparation.