STATISTICAL DEBUGGING AND AUTOMATED PROGRAM FAILURE TRIAGE

BY

CHAO LIU

B.S., Peking University, 2003
M.S., University of Illinois at Urbana-Champaign, 2005

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois
Abstract

Recent years have seen great advances in software engineering and programming languages, and more and more time is devoted to extensive testing and exhaustive debugging. Unfortunately, software is still far from bug-free, even for those deployed. Static analysis is a quality approach to eliminating numerous bugs, but its conservative nature of analysis unavoidably constrains its capacity. Dynamic analysis, on the other hand, utilizes program runtime execution data, and automatically infers about likely problems with the programs, which complements static approaches in ensuring program quality. This thesis describes three dynamic techniques that leverage program runtime data to improve software quality.

First, we present a statistical debugging algorithm, called Sober, which automatically localizes software faults without any prior knowledge of the program semantics. This statistical debugging alleviates the high cost associated with human efforts in manual debugging. Featuring a similar rationale to hypothesis testing, Sober quantifies the fault relevance of each predicate in a principled way. We systematically evaluate Sober under the same setting as previous studies, and compare Sober with other seven algorithms. The result clearly demonstrates the effectiveness: Sober could help developers locate 68 out of the 130 faults in the Siemens suite by examining no more than 10% of the code, whereas the Cause Transition approach proposed by Holger et al. and the statistical approach by Liblit et al. locate 34 and 52 faults, respectively. Moreover, the effectiveness of Sober is also evaluated in an “imperfect world”, where the test suite is either inadequate or only partially labeled. Our experiments indicate that Sober could achieve competitive quality under these harsh circumstances. Finally, four case studies with flex-2.4.7, grep-2.2, gzip-1.2.3, and bc-1.06 are reported, which shed light on the applicability of Sober on reasonably large programs.

Second, we discuss automated program failure triage, which is a closely related problem with automated debugging. Recent software systems usually feature an automated failure reporting
component, with which a huge number of failures are collected from software end-users. In order to effectively leverage the valuable program failure data, the collected failures need to be first triaged, i.e., to locate the most severe failures, and to assign them to appropriate developers. Lying in the center of failure triage is failure indexing, which tries to group failures due to the fault together. Previous studies index program failures based on the trace similarity by hypothesizing that similar failing traces imply the same fault. But because a fault can be triggered in a number of different ways, failing traces due to the same fault can be quite different. In this thesis, we propose a statistical debugging-based approach to program failure triage, called R-PROXIMITY, which better indexes failures and facilitates failure assignment. Two detailed case studies with grep-2.2 and gzip-1.2.3 are provided, which demonstrate the claimed advantages.

Finally, we describe a program dynamic slicing-based approach to failure indexing, which complements R-PROXIMITY. According to our evaluation, R-PROXIMITY is a quality failure indexing tool, but its effectiveness relies on a sufficient number of correct executions, which may or may not be available in practice. The proposed approach based on dynamic slices does not require any correct executions, and hence perfectly complements R-PROXIMITY. Three case studies with grep, gzip, and flex are performed, which validates the advantages of the proposed approach.
To my parents and Juan for your love and encouragement.
Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Jiawei Han, for the lasting support, patience and guidance that he has brought to this project. I became one of his students when he lectured a short-term course on Data Mining in Peking University, summer 2002; and I have been honored to be his student ever since.

The idea of statistical model-based software bug localization could not have been possible without the contributions from fellow researchers. Especially, I would like to thank Xifeng Yan: it was the intensive discussion with him that brought up the idea. My Ph.D. greatly benefited from the numerous dynamic and stimulating discussion with Prof. Darko Marinov, who taught me how to behave positively and work effectively under pressure. I am also grateful to our responsible collaborators, Prof. Sam P. Midkiff and his student Long Fei, and Prof. Xiangyu Zhang in the Purdue University. I also would like to thank Prof. Gregg Rothermel generously shared the Siemens Programs suite with us, and the GrammaTech Inc. for its free copy of the CODESURFER software, which constitute the cornerstone of my experimental evaluation.

Finally, I am indebted to my parents Baishun Liu and Yueping Qiu, and my wife Juan Wang for their continuous support and encouragement. It is the love they have instilled in me that has been motivating my academic pursuits and careers.
Table of Contents

List of Tables ......................................................... x

List of Figures ......................................................... xi

Chapter 1 Introduction .............................................. 1
  1.1 Motivation .................................................. 1
  1.2 Statistical Debugging ....................................... 3
  1.3 Automated Program Failure Triage ......................... 4
  1.4 Thesis Contribution ......................................... 5

Chapter 2 Related Work ............................................. 7
  2.1 Automated Fault Localization ............................... 7
    2.1.1 Fault Localization based on Static Program Analysis .. 7
    2.1.2 Fault Localization based on Dynamic Program Analysis .. 9
  2.2 Program Failure Triage .................................... 12

Chapter 3 SOBER: Statistical Model-Based Fault Localization .. 15
  3.1 Introduction ................................................ 15
  3.2 A Motivating Example ...................................... 18
  3.3 Predicate Ranking Models ................................... 21
    3.3.1 Problem Settings ..................................... 21
    3.3.2 Probabilistic Treatment of Predicates ................ 21
    3.3.3 Methodology Overview ................................ 22
    3.3.4 Predicate Ranking ..................................... 23
    3.3.5 Discussions on Score Computation ..................... 25
    3.3.6 Generalizing Invariants ................................ 26
    3.3.7 Differences between SOBER and Liblit05 ............... 28
  3.4 Empirical Comparison with Existing Techniques .......... 31
    3.4.1 Experimental Setup .................................... 31
    3.4.2 Compared Fault Localization Algorithms ............... 34
    3.4.3 Comparison with Liblit05 and CT ...................... 36
    3.4.4 Comparison with Tarantula and SliceChop ............. 39
    3.4.5 Comparison from Other Perspectives ................... 41
  3.5 SOBER in an Imperfect World ............................... 42
    3.5.1 Robustness to Inadequate Test Suites ................ 43
    3.5.2 Handling Partially labelled Test Suites ............... 44
    3.5.3 Summary ................................................ 48
3.6 Experimental Evaluation with Large Programs ........................................ 48
  3.6.1 Subject Programs and Seeded Faults ........................................... 48
  3.6.2 An Overview of Debugging Quality and Comparison ........................... 50
  3.6.3 A Detailed Case Study with Grep-2.2 .......................................... 52
  3.6.4 Effectiveness with Memory Faults: A Case Study with Bc 1.06 ............... 54
  3.6.5 Summary .................................................................................. 56
3.7 Discussion ......................................................................................... 57
  3.7.1 Related Work ........................................................................... 57
  3.7.2 Threats to Validity ...................................................................... 58
3.8 Conclusions ....................................................................................... 59

Chapter 4 Statistical Debugging-Aided Failure Triage ................................. 60
  4.1 Introduction ...................................................................................... 60
  4.2 Preliminaries and Terminologies ...................................................... 64
  4.3 R-Proximity: Failure Proximity Defined via Statistical Debugging ............ 65
    4.3.1 Fault-aware Fingerprinting ....................................................... 65
    4.3.2 Classic Kendall’s tau Distance ................................................. 67
    4.3.3 Weighted Kendall’s tau Distance .............................................. 68
  4.4 Advantages of R-Proximity .............................................................. 71
    4.4.1 Fault-aware Failure Clustering ............................................... 71
    4.4.2 Guided Failure Assignment .................................................... 72
    4.4.3 Interpretation of Debugging Results ...................................... 73
  4.5 Advantages Illustrated with Case Studies .......................................... 74
    4.5.1 Case Study I: Grep-2.2 ........................................................... 74
    4.5.2 Case Study II: Gzip-1.2.3 ....................................................... 81
    4.5.3 Time Efficiency ...................................................................... 85
    4.5.4 Summary ................................................................................ 85
  4.6 Parameter Effects on Failure Clustering ............................................. 86
    4.6.1 Is Predicate Selection Critical? ............................................... 87
    4.6.2 Is Predicate Weighting Critical? ............................................. 87
    4.6.3 Parameter Effects ................................................................... 88
  4.7 Discussions ...................................................................................... 89
    4.7.1 Related Work ......................................................................... 89
    4.7.2 Threats to Validity ................................................................ 90
  4.8 Conclusion ....................................................................................... 91

Chapter 5 Failure Indexing through Program Dynamic Slices ....................... 92
  5.1 Introduction ...................................................................................... 92
  5.2 A Distance Metric-based Framework for Failure Indexing ....................... 96
    5.2.1 Failure Indexing in Formulation ............................................. 96
    5.2.2 An Evaluation Metric ........................................................... 98
  5.3 Dynamic Slicing-based Failure Indexing ............................................ 100
    5.3.1 Dynamic Slices as Failure Signatures ..................................... 100
    5.3.2 Distances between Dynamic Slices ....................................... 103
    5.3.3 Failure Indexing in Visualization ......................................... 104
  5.4 Experiment Result ........................................................................... 105
    5.4.1 Experiment Setup .................................................................. 105
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Characteristics of Subject Programs</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Characteristics of the Three Subject Programs</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Debugging Quality Evaluated on Large Programs</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Top Three Predicates from the First Run of SOBER</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>Selected Predicates for Grep</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Culpability Assignment at the Existence of Two Faults</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>Selected Predicates for gzip-1.2.3</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>Different Indexing Techniques under the same Distance Metric-based Framework</td>
<td>97</td>
</tr>
<tr>
<td>5.2</td>
<td>Characteristics of the Three Subject Programs</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Failure Group Determination of Failure $x$</td>
<td>107</td>
</tr>
</tbody>
</table>
# List of Figures

3.1 Faulty Code - Version 3 of replace .............................................. 18
3.2 Branching Actions in $P$ (left) and $\bar{P}$ (right) .............................................. 19
3.3 A Correct (left) and An Incorrect (right) Execution in $P$ .............................................. 20
3.4 Methodology Difference between Liblit05 and SOBER .............................................. 29
3.5 Located Faults w.r.t. Code Examination .............................................. 36
3.6 Quality Comparison w.r.t. Various top-$k$ Values .............................................. 37
3.7 Quality of SOBER w.r.t. top-$k$ Values .............................................. 38
3.8 Quality Comparison between Existing Algorithms .............................................. 40
3.9 Quality Degradation w.r.t. $\beta$%-Sampled Test Suite .............................................. 41
3.10 Two Schemes to Work with Partial-labelled Test Suite .............................................. 45
3.11 Quality Comparison w.r.t. the Number ($m$) of Labelled Failing Cases .............................................. 45
3.12 Five Faults in Flex-2.4.7 .............................................. 49
3.13 Two Faults in Grep-2.2 .............................................. 50
3.14 Two Faults in Gzip-1.2.3 .............................................. 51
3.15 Visual Comparison of Debugging Quality between SOBER and Liblit05 .............................................. 53
3.16 First Fault in bc 1.06, in storage.c .............................................. 55
3.17 Second Fault in bc 1.06, in storage.c .............................................. 56

4.1 T-Proximity and R-Proximity .............................................. 62
4.2 Rank Projection and Predicate Weighting .............................................. 68
4.3 Two Injected Faults in grep-2.2 .............................................. 75
4.4 Proximity Graphs with R-Proximity (left), the close-up of Cluster2 (middle) and with T-Proximity (right) for grep-2.2 .............................................. 77
4.5 Case Study with grep-2.2: Top-$k$ Spectrum Graphs for Cluster1 .............................................. 78
4.6 Case Study with grep-2.2: Top-$k$ Spectrum Graphs for Cluster2 .............................................. 79
4.7 Proximity Graphs with R-Proximity (left) and with T-Proximity (right) for grep-2.2 .............................................. 81
4.8 Two Injected Faults in gzip-1.2.3 .............................................. 82
4.9 Proximity Graphs with R-Proximity (left) and T-Proximity (right) for gzip-1.2.3 .............................................. 82
4.10 Case Study with gzip-1.2.3: Top-$k$ Spectrum Graphs for Cluster1 .............................................. 83
4.11 Case Study with gzip-1.2.3: Top-$k$ Spectrum Graphs for Cluster2 .............................................. 83
4.12 Colorbar: Correspondence between $T$-scores and Representation Colors in Proximity Graphs .............................................. 86
4.13 Effect of Predicate Selection .............................................. 87
4.14 Effect of Predicate Weighting .............................................. 87
4.15 Grouping of Failing Cases with Different $k_1$’s .............................................. 88
4.16 Grouping of Failing Cases with Different $\alpha$’s .............................................. 88
5.1 An Illustration of Silhouette Coefficients ........................................ 100
5.2 Data Slice (left) and Full Slice (left) ............................................... 101
5.3 Pruned Slice (PS) ................................................................. 103
5.4 Illustration of the Dynamic Slicing System ...................................... 106
5.5 Comparison with T-Proximity and R-Proximity on Grep-2.2 ............... 108
5.6 Two Seeded Faults in Gzip-1.2.3 ............................................... 109
5.7 Comparison with T-Proximity and R-Proximity on Gzip-1.2.3 .......... 110
5.8 Data Slice of the Test Case 8 .................................................... 111
5.9 Data Slice of the Test Case 82 ................................................... 112
5.10 Two Seeded Faults in Flex-2.4.7 .............................................. 113
5.11 Comparison with T-Proximity and R-Proximity on Flex-2.4.7 ........... 114
5.12 Summarized Comparison .......................................................... 115
5.13 Effectiveness of Pruned Slices .................................................. 120
Chapter 1

Introduction

This chapter motivates, and provides general introduction to, the two major problems tackled in this thesis, namely, statistical debugging and automated program failure triage. We summarize the contributions of this thesis at the end of this chapter.

1.1 Motivation

Bug-ridden software remains one of the major headaches in our modern society. However, despite the great advances in software engineering and programming languages as witnessed in the last decade, software is still far from bug-free. Because software development is still primarily a human activity and the rapid growth of software complexity remains unabated, software faults\(^1\) inevitably creep in complex software that is used by billions of people every day. According to a study done by Humphrey in 2004, software on average contains 10 to 20 faults per thousand lines of code after compiling and testing \([43]\). The huge number of hidden bugs constitutes the major source of frustration and fear, for both software vendors and end users. As estimated by the National Institute of Standards and Technology (NIST) in 2002, software bugs caused the US economy an estimated $59.5 billion annually. Therefore, how to improve the quality of software by reducing the number of errors has posed an imminent challenge to the research community.

In order to improve the software quality, both static and dynamic analysis-based approaches have been pursued. Static analysis \([10, 40, 29, 18, 76, 23, 26, 44]\) can sometimes guarantee the program free of certain types of bugs, but its limitations are also great. First, the identified faults through static analysis tend to contain a high rate of false positives, which is because of the conservative nature of the underlying analysis techniques such as alias analysis. Second, static

\(^{1}\)We use bug and fault interchangeably to refer to the incorrect source code, and use failure to refer to the resulting incorrect behavior due to the fault.
analyses are only capable of verifying certain simple properties, like no dereferences to NULL pointers. But more bugs are much trickier than those covered by simple properties. Finally, static analyses usually require developers to provide specifications, which many developers are reluctant to write. Therefore, static analyses cannot remove all the software errors.

When a software bug escapes detections based on static analysis, we would rely on dynamic analysis to catch it at runtime. Basically, dynamic analysis monitors the program runtime behavior, and tries to locate bug through contrasting the runtime behavior of correct and incorrect executions. In contrast to static analysis which often requires certain kind of specification, dynamic techniques do not assume any prior knowledge of program semantics other than the labelling of each execution as either correct or incorrect. Previous studies deploy a variant of program runtime behaviors for fault localization, such as program spectra [41] [81], memory graphs [95] [23], and program predicate evaluation history [58] [59]. In this thesis, we describe a new dynamic analysis-based approach to fault localization, SOBER, which has been shown as one of the most accurate debugging algorithms. We will discuss about it with details in Section 1.2 and Chapter 3.

Besides utilizing runtime execution data from in-house testing, dynamic analysis also leverages execution data collected from the field by virtue of a software practice known as automated failure reporting. Whenever a program fails, failure relevant data is automatically collected, and reported to the central server for analysis. Representative failure reporting systems have been successfully deployed in most complex software systems, such as Microsoft Windows and Mozilla Applications. However, since the sheer number of collected failures easily exceeds the capability of human developers, automated failure triage techniques are needed to fully leverage the valuable failure data collected from the field. Roughly speaking, failure triage contains two major problems: failure prioritization and failure assignment. Previously, Podgurski et al., proposed a triage approach based on program execution traces, which can automate the failure prioritization step. In this thesis, we present two failure triage methods, both of which can automate the two tasks of failure triage. Technical details about our failure triage techniques are provided in Chapters 4 and 5.

The following two sections discuss our approaches to statistical debugging and failure triage, but without delving into technical details. Therefore, they can be treated as previews of Chapters 3, 4 and 5.
1.2 Statistical Debugging

Statistical debugging is to localize faulty source code through statistical analysis of failing and passing executions. The basic idea is to detect runtime abnormality in failing executions through comparison with passing ones. Different from previous work [80, 101] and existing debugging tools, like Valgrind [84] and Purify [42], my study focuses on localizing semantic bugs. In general, semantic bugs are harder to locate than memory bugs for a number of reasons, some of which are:

- Semantic bugs are program-specific while memory bugs are mostly generic. For example, missing a subclause may or may not be a bug, but dereferencing a NULL pointer is always.
- Because of the program-specific failure behavior of semantic bugs, generic debugging policies are hard to develop for semantic bugs. In comparison, the generic symptom of memory bugs renders effective debugging policies possible. For example, a simple validation of every memory access is an effective way to detect memory bugs.
- Semantic bugs can fail a program without crashes, and the unavailability of crashing venues usually renders developers clueless in debugging. In comparison, as most memory bugs crash programs, developers can usually trace back to the root cause from the crashing venue.
- Finally, the inherent complexity of program logic makes it hard to unravel the manifestation of semantic bugs. For example, most semantic bugs incur incorrect control flows and wrong value propagations, which are all invisible symptoms. Due to the invisibility, developers usually have to trace the execution to clear semantic bugs.

A recent study of bug characteristics done by Li et al. [56] validates the above claims with concrete bug statistics. The authors manually examine 264 and 98 bugs in Mozilla and Apache Server, respectively, and find that semantic bugs

- account for 81.1-86.7% of all the examined bugs, and this percentage increases with the software maturity.
- have severe impacts on system availability, contributing to 42.9-44.2% of program crashes.
- take almost twice as much as time to debug than memory bugs.
also dominate the cause of security problems, accounting for 71.9-83.9% of all examined security problems. Buffer overflows are no longer the major cause because of quality memory checking tools.

These statistics testify that as more and more memory bugs can be found by effective memory checking tools, noncrashing failures that are due to semantic bugs have become dominant. More generally, most annoying problems in computer systems are also noncrashing failures due to semantic bugs, such as misconfigurations and spyware exploitation. Therefore, debugging semantic bugs is an important practical problem to solve. Specifically, since specification-based approach to checking semantic bugs does not scale with the variety and fast changing pace of computer systems and programs, we believe that using statistical methods to automatically untangle program semantics is definitely a viable and promising approach.

With such a belief in mind, we have developed a statistical debugging algorithm, called SOBER, which automatically localizes faulty source code that cause program noncrashing failures through statistical analysis of execution data. The basic idea is to statistically model the passing and failing executions, and then mathematically quantify how failing executions diverge from passing executions. The divergence place suggests the likely bug locations.

Because SOBER’s analysis does not rely on crashing venues, it is applicable to both semantic and memory bugs. We compare SOBER with other 7 existing algorithms on a set of semantic bugs and illustrate its effectiveness with case studies of 11 faults on four median-sized programs. Especially, SOBER helps us find a memory bug in bc, which has not been reported before. We will delve into the details of SOBER in Chapter 3.

1.3 Automated Program Failure Triage

Automated program failure triage is a problem that arises from the popularity of automated failure reporting components, which are widely distributed in deployed software. On program failures (either crashing or noncrashing), such components automatically collect failure-relevant information and report (with the user’s permission) such information to software vendors for failure patches. However, as the sheer number of reported failures easily exceeds the capability of manual investigation, failures need to be triaged. Specifically, failure triage includes two problems: failure
prioritization and failure assignment.

Failure prioritization is to prioritize reported failures so that the most severe failures are diagnosed first by developers. Usually, the failure severity is determined by the number of reports for the failure. As failures due to the same bug can be reported from different users in different forms, clustering is a common approach to identifying failures due to the same bug. Ideally, failures due to the same bug are expected to be clustered together while being separated from failures due to other bugs.

Failure assignment, on the other hand, asks another related question: Now that this cluster of failures should be diagnosed first, who are the most appropriate developers to diagnose it? To the best of our knowledge, failure assignment has been mainly manual in practice.

Failure triage can be straightforward for crashing failures because the same crashing venue is a strong indicator of failures due to the same bug. Therefore, the hard side of failure triage is for noncrashing failures, which are related to semantic bugs. As semantic bugs become dominant (and so do noncrashing failures), triage algorithms for noncrashing failures are needed.

In Section 2.2, we survey current approaches to failure triage, and identify their shortcomings in handling noncrashing failures. In Chapter 4, we present our statistical debugging-aided approach, which addresses failure prioritization and assignment simultaneously.

While the statistical debugging-based approach is every effective in triaging noncrashing failures, its applicability relies on the availability of a set of correct executions. Unfortunately, this availability cannot be safely assumed in practice, for example, because the transmission of a sufficient set of correct executions will nontrivially hamper the network usage. Therefore, we propose a dynamic slicing-based approach to failure triage, which complements the statistical debugging-aided approach when correct executions are not available or too few to rely on. By taking dynamic slices as failure signatures, we can obtain comparable result for failure triage. Details about this dynamic slicing-based approach is discussed in Chapter 5.

1.4 Thesis Contribution

In summary, this thesis makes the following contributions

- We presented a new statistical debugging algorithm SOBER, which features a rigid math-
ematical foundation, not ever seen in previous work. Especially, we showed that SOBER encompasses invariant-based bug detection techniques as a special case.

• We systematically evaluated SOBER using the same evaluation framework as previous work, and compared SOBER with other 7 debugging algorithms, which claimed or had claimed the best result. The evaluation results clearly demonstrated the effectiveness of SOBER. In order to further illustrate how SOBER can help developers locate software bugs, four case studies with median-sized programs were also performed, which also shed some light on the applicability of SOBER on reasonably large programs.

• We formulated the problem of failure triage, and proposed a statistical debugging-aided approach, called R-PROXIMITY, which indexes failures and facilitates failure assignment better than previous work. Also, for the first time, we showed that fault localization algorithms are not restricted to debug software programs; instead they can be leveraged for various dynamic analysis problems in general.

• We further investigated the problem of failure triage, and proposed a dynamic slicing-based technique to indexing program failures without reliance on any correct executions. This technique perfect complements the statistical debugging-aided approach when no or only insufficient number of correct executions are available.

The rest of this thesis is organized as follows. Chapter 2 presents a comprehensive and in-depth survey of related work for both automated fault localization and failure triage. Chapter 3 discusses at length the statistical debugging algorithm SOBER, and compares it with other 7 existing debugging algorithms. With the development of SOBER, Chapter 4 elaborates on the statistical debugging-aided approach to automated program failure triage. The dynamic slicing-based techniques to failure indexing are provided in Chapter 5, and finally Chapter 6 concludes the thesis.
Chapter 2

Related Work

In this chapter, I review related work to fault localization and failure triage in Sections 2.1 and 2.2, respectively. Especially, since automated fault localization can be based on either static or dynamic program analysis, two categories are discussed separately within Section 2.1.

2.1 Automated Fault Localization

2.1.1 Fault Localization based on Static Program Analysis

Fault localization can be based on static program analysis, where program source code is checked against a program correctness model. Depending on how the correctness model is constructed and how formal the checking is, a whole spectrum of algorithms have been studied before.

The simplest checking is usually implemented through regular compilers, which can detect obvious but frequently made programming errors, such as syntax errors and type violations. For this type of checking, the correctness model is embedded into the language specification and type systems. On the other extreme is program verification, where formal methods are employed to demonstrate that a program does implement a desired specification [9, 22]. Program verification can guarantee the absence of specific errors by searching all possible program executions, and hence is usually sound. But the associated problem with soundness is false positives and scalability, which, together with requirement of manually specified correctness model, constitute the major barriers to the widespread adoption of program verification.

Sitting between these two extremes are specialized checkers, which could be neither sound nor complete. But as they are designed for specialized purposes, they are usually scalable enough for practical deployment. The LCLint tool is a light-weighted static checker [36]. It takes any ANSI C programs as inputs, and checks for any inconsistencies between the code and LCL specifications.
The LCL interface specification language provides users with different levels of specifications to suit users’ different needs. In a subsequent work [35], the author extends the LCL interface specification language to annotate assumptions about memory allocation, initialization and sharing, so that dynamic memory errors can be detected. Constraints derived from annotations are checked at compile time, and any violations are considered as potential errors. PREfix [19] is another important work in static analysis-based fault localization. It focuses on detecting problems due to function interactions through inter-procedural analysis. Specifically, it traverses a program’s call graph in a bottom-up fashion, and builds a model for each function upwards. The model for each function consists of a set of paths that likely invoke erroneous behaviors, for example, returning a NULL pointer. When building the model for a caller function, PREfix checks whether the model of the callee function can incur program errors, for example, whether the caller function will dereference a NULL pointer that is possibly returned by the callee. Because PREfix is essentially heuristic, its analysis is scalable. It has been widely deployed in Microsoft, and successful experience has been reported [53].

Going beyond analyzing the interaction between functions, a set of tools have been developed to check for event ordering problems. The SLAM tool [12] models correct event sequence rules through finite state machines (FSM), and checks a C program to either find plausible program paths that violate the rule or determine that all paths respect the rule. It simplifies the analysis and scales to large problems by abstracting a program into a Boolean program, eliminating irrelevant details. It has been shown especially useful for checking device drivers. BLAST [86] is a similar technique, but focuses on checking safety related properties. One more similar tool to SLAM is the ESP tool [29], which further scale SLAM to very large programs at a loss of precision. Especially, it lets users to specify rules in the simple Object Property Automata Language (OPAL), which combines an FSM with syntactic code patterns. As the OPAL rules relates FSM to program syntax, the checking can be performed in a grep-like fashion.

Although static analysis features a number of nice properties, one shortcoming is its reliance on the specification that usually has to be specified manually. Therefore, another line of research has been devoted to mining specifications from program data automatically [7]. Engler et al. propose to infer programmers’ beliefs from source code automatically [33]. Such beliefs are then
cross-checked for contradictions, which indicate program errors. While Engler et al. discover beliefs according to human written templates, Li et al. adopt a more systematic approach to discovering programming rules by exploiting data mining techniques. Specifically, their CP-Miner uses CloSpan [94] to discover closed frequent subsequential patterns from source code, and relates inconsistencies to copy-paste errors [55]. Similarly, their subsequent work on PR-Miner automatically finds from source code programming rules (PR), rules that developers are expected to obey, and warns about errors for rule violations [57]. Besides source code, other kinds of program data are also good source for specification mining. For example, Ammons et al. discover formal specifications from executions, a form of dynamic program data [7]. And Livshits and Zimmermann demonstrate that software revision histories can be mined for common error patterns [65].

Finally, we note that although static analysis and dynamic analysis are discussed separately here, they are not dichotomic in practice. In fact, an elegant combination of them can usually generate promising results. For example, Manevich et al. propose using postmortem static analysis to locate faults [67]. The idea is that once information about dynamic program failure is known (e.g., failure types and failure location), static analysis becomes more directed so that fault localization is more accurate than pure static analysis-based approach. I envision that the interaction between static and dynamic analysis, as well as the utilization of data mining and machine learning algorithm, will make fault localization more accurate and flexible.

2.1.2 Fault Localization based on Dynamic Program Analysis

Fault localization techniques based on dynamic program analysis reason about program faults from program real executions, in comparison with simulated executions in static analysis. By virtue of the access to real executions, dynamic analysis does not suffer from the same conservativeness problem as static analysis does. Basically, dynamic analysis instruments subject programs first, and tries to identify abnormal behaviors at program runtime; and finally relates abnormal behaviors to bug locations.

The abnormal behaviors can sometime be very salient, e.g., dereferencing to a NULL pointer. Dynamic analysis can detect such faults by consistently monitoring memory accesses. Some tools, like Valgrind [84] and Purify [42], are very effective in detecting faults associated with memory access
violations. While effective, these tools usually bear very high runtime overhead (e.g., 20X slowdown for Purify), which motivates substantial researches on lowering down the overhead [101, 80].

On the other hand, when the abnormal behavior is not apparent, e.g., a program gives the wrong output without crash, fault localization needs to contrast the erroneous executions to some correct executions to find the abnormal behaviors, and further relate them to potential fault locations. While semantic bugs usually fall into this category, so are tricky memory bugs which do incur crashes. A number of previous studies on fault localization [58, 59, 64, 60] indicate that the crashing point can be far from the real fault location. Therefore, in-depth analysis is needed to trace down the real fault location.

Previously, an invariant-based approach has been investigated. The basic idea is to learn invariant properties that hold in program correct (or assumed correct) executions so that any future invariant violations can be related to program faults. Ernst et al. first introduce and formalize the notion of invariants in the DAIKON project, and propose a relaxing-based approach to discover invariants automatically from program executions [34]. Later, the authors exemplify the usage of invariants in fault localization [18, 69]. The DIDUCE project [39] monitors a more restricted set of predicates and relaxes them in a similar manner to DAIKON at runtime. After the set of predicates becomes stable, the DIDUCE tool relates further violations as indications of potential faults. This approach is shown effective on tracking down problems in four large software systems. Zhou et al. extend the invariant idea to program counter-based invariants, and relate invariant violations to memory related bugs. However, as invariants are nevertheless a special kind of predicates that hold in all passing runs, they are not effective in locating subtle faults as suggested by Pytlik and Renieris in [79]. In comparison, the probabilistic treatment of predicates implemented by SOBER naturally relaxes this requirement and is shown to achieve much better localization results on the Siemens suite.

Zeller and his colleagues present a series of fault localization techniques based on the memory graph presentation [102] of program execution states. In a memory graph, all values and all variables are represented as vertices, and operations like variable access, pointer dereferencing, struct member access, or array element access as edges. Zeller and Hildebrandt propose an algorithm to simplify and isolate failure-inducing input by systematically narrowing down the “delta” input that
determines the occurrence of program failures [96]. Building on the simplified “delta” input, Zeller proposes the Delta Debugging algorithm, which isolates the relevant variables and values by systematically narrowing the state difference between a passing execution and a failing one. While the state to be compared needs to be pre-selected for Delta Debugging, Cleve and Zeller later propose the Ct algorithm that automates the search for appropriate states to compare. Although as will be shown in Section 3.4, Delta Debugging is no longer the most effective algorithm, its methodology has been influential for fault localization research. Even recently, numerous work has been built on top of Delta Debugging, and shows quality result, e.g., the Hierarchical Delta Debugging algorithm by Jiang and Su [72] and the SLICECHOP algorithm by Gupta et al. [38].

Because program dependencies are natural resource to consider for debugging, algorithms based on contrasting program slices are also pursued and shown effective for fault localization [51, 87, 5, 48, 13, 6, 77]. Agrawal et al. [6] present a fault localization technique, implemented as $\chi$slice, which subtracts a single correct execution trace from a single failed execution trace. In [77], Pan and Spafford presented a family of heuristics for fault localization using dynamic slicing. Jones et al. [46] describe a similar approach implemented as TARANTULA. Unlike $\chi$slice, TARANTULA collects the testing information from all passing and failing cases, and colors suspicious statements based on the contrast. Later, Renieris and Reiss [81] find that the contrast renders better fault localization when the given failing case is contrasted with the most similar passing case (i.e., the nearest neighbor). Gupta et al. recently propose a fault localization algorithm SLICECHOP that integrates Delta Debugging and dynamic slicing [38]. The idea of SLICECHOP is to first find a simplified failure-inducing input $f'$ from the given failing case $f$ using Zeller and Hildebrandt’s algorithm [96]. Then a forward dynamic slice $FS$, and a backward slice $BS$ are calculated from $f'$ and the erroneous output, respectively. Finally, the intersection of $FS$ and $BS$, i.e., the chop, is taken as the fault localization report, namely, $R = FS \cap BS$. As expected, SLICECHOP outperforms techniques based on either Delta Debugging or dynamic slicing.

Liblit et al. propose statistical debugging, which instruments programs with predicates and localizes program faults through a statistical analysis of predicate profiles of executions, both correct and incorrect ones. They describe a sampling framework and present an approach to guess and eliminate predicates to isolate a deterministic bug [58]. For isolating nondeterministic bugs, they
use logistic regression techniques to identify predicates that are highly correlated with the program failure. Later, the authors propose several heuristics to prune irrelevant predicates, and achieve better debugging results [59]. Recently, they demonstrate that multiple bugs can be identified simultaneously through bi-clusterings [100]. In Chapter 3, we will present our statistical model-based debugging algorithm Sober, which is shown more accurate. Fei et al. extend the statistical debugging idea to online debugging, in which a correct model is built by training through a set of correct runs, and any significant deviation from this model in the detection run raises a flag [37].

While all the fault localization algorithms examined in this article are designed for programming professionals, recent years have also witnessed an emergence of fault localization algorithms especially tuned to assist end users in fault diagnosis. For example, Ayalew and Mittermeir propose a technique to trace faults in spreadsheets based on “interval testing” and slicing [10]. Ruthruff et al. improve this approach by allowing end-users to interactively adjust their feedbacks [83]. The Whyline prototype realizes a new debugging paradigm called “interrogative debugging”, which allows users to ask why did and why didn’t questions about runtime failures [50].

### 2.2 Program Failure Triage

Failure triage becomes a critical research problem because of the increasing popularity of bug tracking systems [70]. A bug tracking system keeps record of reported failures, and supports bug diagnosis and software evolution. Depending on the designed functionality, bug tracking systems can support either manual or automated failure reporting. In either way, the reported failures need to be triaged so that reported failures are prioritized and assigned to appropriate developers for patches. Therefore, failure triage includes two important tasks: failure prioritization based on severity and failure assignment.

Some bug tracking systems are designed for interaction with developers and technically savvy software users. For example, the Bugzilla system, which most open-source projects rely on for bug tracking, falls into this category. To report a failure in such system, the reporter needs to specify numerous technical details, such as execution environment, failure stack trace on crashes and how to reproduce. In such systems, it is usually human triagers that decide the priority of reported failures, and assign failures to the appropriate developers. Because such manual processing cannot
scale with the increasing number of failure reports that come from the growing size and complexity of software, automated assistance has been pursued recently. Cubranic and Murphy propose a text categorization-based approach to failure assignment [28], whose main idea is to predicate the appropriate developers for a new failure report by using a text classifier trained from past records of failure assignment. Later, Anvik et al. consolidate this text categorization-based approach, and present an empirical study with open-source projects Eclipse and Firefox [8]. The empirical study shows that the predication of failure assignment is less accurate (50% precision) so that the proposed approach only semi-automates failure assignment. In comparison with our discussion in Chapter 4, the above work does not address the problem of failure prioritization.

On the other hand, some bug tracking systems aim at collecting and fixing failures experienced by end-users in production runs. Typical representatives are the Dr. Watson System by Microsoft Corporation and the Cooperative Bug Isolation (CBI) project. As end-users are not necessarily technically savvy enough, these systems save users’ hassles in providing intricate failure details. The Dr. Watson System basically collects failure venues (e.g., call stack traces), and a CBI’s failure report contains the predicate profiling of executions. For these systems, failure triage is performed by clustering failures likely due to the same fault together. To the best of our knowledge, the Dr. Watson System cluster failures with the same failure venue together. Although for most cases stack traces are a good indicator of the underlying fault, they are not always effective as indicated in [60], and especially when no crashes take place. For these cases, Podgurski et al. propose to cluster failure reports according to the execution trace similarity [78], and show that failures in the same cluster likely correspond to the same fault. However, as will be shown in Chapter 4, because this approach hypothesize that the same fault always incur failure in a similar way, which may not necessarily hold, the cluster results are quite fuzzy. Moreover, while the clustering result can be used for failure prioritization, an extension to tackle the problem of failure assignment is not straightforward. In Chapter 4, we propose a statistical debugging approach which solves the failure prioritization and assignment problems simultaneously.

In recent study, Tucek et al. propose another meaning of failure triage [88]. The authors present a Triage Diagnosis Protocol (TDP) to diagnose program failures at user-side automatically. The protocol is basically a recipe for online failure diagnosis. Because it focuses more on failure
diagnosis rather than prioritization and assignment, their usage of the term “triage” disagrees with the traditional definition.
3.1 Introduction

The last decade has witnessed great advances in fault localization techniques [22, 90, 75, 81, 95, 23, 58, 47, 38]. These techniques aim to assist developers in finding fault locations, which is one of the most expensive debugging activities [89]. Fault localization techniques can be roughly classified as static or dynamic. A static analysis detects program defects by checking the source codes with or without referring to a well-specified program model [22] [90] [75]. A dynamic analysis, on the other hand, typically tries to locate defects by contrasting the runtime behavior of correct and incorrect executions. Dynamic techniques usually do not assume any prior knowledge of program semantics other than the labelling of each execution as either correct or incorrect. Previous studies deploy a variant of program runtime behaviors for fault localization, such as program spectra [41] [81], memory graphs [95] [23], and program predicate evaluation history [58] [59].

Within dynamic analyses, techniques based on predicate evaluations have been shown to be promising for fault localization [18] [39] [58] [59]. Programs are first instrumented with predicates such that the runtime behavior in each execution is encoded through predicate evaluations. Consider the predicate “\( \text{idx} < \text{LENGTH} \)”, where the variable \( \text{idx} \) is an index into a buffer of length \( \text{LENGTH} \). This predicate checks whether accesses to the buffer ever exceed the upper bound. Statistics on the evaluations of predicates are collected over multiple executions at runtime and analyzed afterwards.

The method described in this article shares the principle of predicate-based dynamic analysis. However, by exploring detailed statistics about predicate evaluation, our method can detect more and subtler faults than the state-of-the-art statistical debugging approach proposed by Liblit et al. [59]. For easy reference, we denote this method as LIBLIT05. For each predicate \( P \) in a program...
Liblit05 estimates two conditional probabilities: \( Pr_1 = Pr(\mathcal{P} \text{ fails}|\mathcal{P} \text{ is ever observed}) \) and \( Pr_2 = Pr(\mathcal{P} \text{ fails}|\mathcal{P} \text{ is ever observed as true}) \). It then treats the probability difference \( Pr_2 - Pr_1 \) as an indicator of how relevant \( \mathcal{P} \) is to the fault. Therefore, Liblit05 essentially regards a predicate fault-relevant if its true evaluation correlates with program failures.

While Liblit05 succeeded in isolating faults in some widely used software [59], it has a potential problem in its ranking model. Because Liblit05 only considers whether a predicate has ever been evaluated as true or not in each execution, it loses its power to discriminate when a predicate \( \mathcal{P} \) is observed as true at least once in all executions. In this case, \( Pr_1 \) is equal to \( Pr_2 \), which suggests that the predicate \( \mathcal{P} \) has no relevance to the fault. In Section 3.2, we will present an example where the most fault-relevant predicate reveals only a small difference between \( Pr_1 \) and \( Pr_2 \). We found that similar cases are not rare in practice, as suggested by the experiments in Section 3.4.

The above issue motivates us to develop a new approach that can exploit multiple evaluations of a predicate within each execution. We start by treating the evaluations of a predicate \( \mathcal{P} \) as independent Bernoulli trials: Each evaluation of \( \mathcal{P} \) gives either true or false. We then estimate the probability of \( \mathcal{P} \) being true in each execution, which we call the evaluation bias. While the evaluation bias of \( \mathcal{P} \) may fluctuate from one execution to another, its observed values from multiple executions constitute a random sample from a statistical model. Specifically, if we let \( X \) be the random variable standing for the evaluation bias of predicate \( \mathcal{P} \), then there are two statistical models, \( f_P(X|Correct) \) and \( f_P(X|Incorrect) \), which govern the evaluation bias observed from correct and incorrect executions respectively. Intuitively, if the model \( f_P(X|Incorrect) \) is significantly different from \( f_P(X|Correct) \), it is indicated that \( \mathcal{P} \)'s evaluation in incorrect runs captures abnormal activity, and the predicate \( \mathcal{P} \) is likely relevant to the fault. Therefore, instead of selecting predicates correlated with program failures as done by Liblit05, our approach statistically models predicate evaluations in both correct and incorrect runs respectively and treats the model difference as a measure of the fault relevance.

In quantifying the model difference between \( f_P(X|Correct) \) and \( f_P(X|Incorrect) \), there are two major obstacles. First, we have no idea what family of distributions the two models are in. Secondly, we are not authorized to impose model assumptions on \( f_P(X) \) because improper model assumptions can result in misleading inferences [20]. Therefore, without prior knowledge of the
statistical models, a direct measurement of the model divergence is difficult, if not fully impossible.

In this article, we propose a hypothesis testing-like approach, which indirectly quantifies the model difference. Aiming at the model difference, we instead propose the null hypothesis that the two models are identical. We then derive a statistic that conforms to a normal distribution under the null hypothesis through the Central Limit Theorem. Finally, given observed evaluation biases from multiple executions (both correct and incorrect), the instantiated statistic quantifies the likelihood that the evaluation biases observed from incorrect runs were generated as if from \( f_P(X|\text{Correct}) \). Therefore, a smaller likelihood suggests a larger discrepancy between the two models, and hence a greater likelihood that the predicate \( P \) is fault-relevant. Using this quantification, we can rank all the instrumented predicates, getting a ranked list of suspicious predicates. Developers can then examine the list from the top down in debugging.

In summary, we make the following contributions in this article:

1. We propose a probabilistic treatment of program predicates that models how a predicate is evaluated within each execution, which exploits more detailed information than previous methods [58] [59]. In addition, this probabilistic treatment naturally encompasses the concept of program invariants [34] as a special case.

2. On top of the probabilistic treatment of predicates, we develop a theoretically well-motivated ranking algorithm, SOBER, that ranks predicates according to how abnormally each predicate evaluates in incorrect executions. Intuitively, the more abnormal the evaluations, the more likely the predicate is fault-relevant.

3. We systematically evaluate the effectiveness of SOBER on the Siemens suite [44] [82] under the same setting as previous studies. Seven existing fault localization techniques are compared with SOBER in this study, which demonstrates the superior accuracy achieved by SOBER in fault localization. Furthermore, the effectiveness of SOBER is also evaluated in an “imperfect world”, where the test suite is either inadequate or partially labelled. The experiment result shows that SOBER is statistically robust to these circumstances.

4. Finally, two case studies with grep and bc are reported, which illustrates the applicability of SOBER on reasonably large programs. In particular, a previously unreported fault is found in
bc, based on the fault localization result from SOBER.

The rest of the article is organized as follows. Section 3.2 first presents a motivating example, which illustrates the advantages of modelling predicate evaluations within each execution. We elaborate on the statistical model, ranking algorithm and its relationship with program invariants in Sections 3.3. An extensive comparison between SOBER and existing techniques is presented in Section 3.4, followed by the evaluation of SOBER in an “imperfect world” in Section 3.5. The two case studies with grep and bc are reported in Section 3.6. With related work and threats to validity discussed in Section 5.5, Section 3.8 concludes this study.

3.2 A Motivating Example

In this section, we present a detailed example that illustrates the advantage of modeling predicates in a probabilistic way. This example inspires us to locate faults by quantifying the divergence between the models of correct and incorrect executions.

```c
01 void subline (char *lin, char *pat, char *sub)
02 {
03     ...
04     while(lin[i] != '\0')
05         m = amatch(lin, 1, pat, 0);
06     if(m >= 0) /* && (lastm != m) */
07         {
08             putsub(lin, 1, m, sub);
09             lastm = m;
10         }
11     ...
12 }
13 }
```

Figure 3.1: Faulty Code - Version 3 of replace

The program in Fig. 3.1 is excerpted from the third faulty version of the replace program in the Siemens suite. The program replace has 507 lines of C code (LOC) and it performs regular expression matching and substitutions. The second subclause in Line 7 was intentionally commented out by the Siemens researchers to simulate a type of fault that may sneak in if the developer fails to think fully about the if condition. Since this is essentially a logic error that does not incur program crashes, even experienced developers would have to use a conventional debugger for step-by-step tracing. Our question is: *Can we guide developers to the faulty location or its vicinity by contrasting the runtime behaviors between correct and incorrect executions?*
For clarity in what follows, we denote the program with the subclause \((\text{lastm} \neq m)\) commented out as the incorrect (or faulty) program \(P\), and the one with the subclause \((i.e., \ (\text{lastm} \neq m)\) is not commented out) as the correct program \(\hat{P}\). Because \(\hat{P}\) is certainly not available when \(P\) is debugged, \(\hat{P}\) is used here to illustrate how our method is motivated. As shown in Section 3.3, our method only collects statistics from the faulty program \(P\), not from \(\hat{P}\).

We declare two boolean variables \(A\) and \(B\) as follows.

\[
A = (m >= 0);
B = (\text{lastm} \neq m);
\]

Let us consider the four possible evaluation combinations of \(A\) and \(B\), and their corresponding branching actions (either \(\text{enter}\) or \(\text{skip}\) the block from lines 8-11) in both \(P\) and \(\hat{P}\). Fig. 3.2 explicitly lists the actions in \(P\) (left) and \(\hat{P}\) (right). The left panel shows the actual actions taken in the faulty program \(P\), while the right one lists the expected actions in \(\hat{P}\).

\begin{align*}
&\begin{array}{c|c|c}
A & \neg A \\
B & \text{enter} & \text{skip} \\
\neg B & \text{enter} & \text{skip} \\
\end{array} & \begin{array}{c|c|c}
A & \neg A \\
B & \text{enter} & \text{skip} \\
\neg B & \text{skip} & \text{skip} \\
\end{array}
\end{align*}

Figure 3.2: Branching Actions in \(P\) (left) and \(\hat{P}\) (right).

Differences between the above two tables reveal that in the faulty program \(P\), unexpected actions take place if and only if \(A \land \neg B\) evaluates to \text{true}. Explicitly, when \(A \land \neg B\) is \text{true}, the control flow actually enters the block, whereas it is expected to skip the block if the logic was correct. This incorrect control flow will likely lead to incorrect outputs. Therefore, for the faulty program \(P\), an execution is incorrect if and only if there exist \text{true} evaluations of \(A \land \neg B\) at Line 7; otherwise, the execution is correct even though the program contains a fault.

While the predicate \(P : (A \land \neg B) = \text{true}\) precisely characterizes the scenario under which incorrect executions take place, there is little chance for any fault locator to spot \(P\) as fault-relevant. The obvious reason is that while we are debugging \(P\), \(\hat{P}\) is not available. Therefore, we have no idea of what \(B\) is, let alone its combination with \(A\). On the other hand, because the evaluation of \(A\) is observable in \(P\), we are interested in whether the evaluation of \(A\) can actually point to the fault. Explicitly, if the evaluation of \(A\) in incorrect executions significantly diverges
from that in correct ones, the if statement at line 7 may be regarded as fault-relevant, which
exactly points to the fault location.

\[
\begin{array}{c|cc|cc}
 & A & \neg A \\
 B & n_{AB} & n'_{AB} & n_{AB} \\
 \neg B & n_{AB} = 0 & n_{AB} \\
\end{array}
\quad
\begin{array}{c|cc|cc}
 & A & \neg A \\
 B & n'_{AB} & n'_{AB} & n_{AB} \\
 \neg B & n'_{AB} \geq 1 & n_{AB} \\
\end{array}
\]

Figure 3.3: A Correct (left) and An Incorrect (right) Execution in P

We therefore contrast how A is evaluated differently in correct and incorrect executions of
P. Fig. 3.3 shows the number of true evaluations for the four combinations of A and B in one
correct (left) and one incorrect (right) execution. The major difference between the two is that
in a correct run, \(A \land \neg B\) never evaluates true \((n_{AB} = 0)\) while \(n'_{AB}\) must be nonzero for an
execution to be incorrect. Since the true evaluation of \(A \land \neg B\) implies \(A = true\), we expect
that the probability for A to be true is different in correct and incorrect executions. In running
5,542 test cases, the true evaluation probability is 0.2952 in a correct execution and 0.9024 in
an incorrect execution, on average. This divergence suggests that the fault location (i.e., Line 7)
does exhibit detectable abnormal behaviors in incorrect executions. Our method, as described in
Section 3.3, nicely captures this divergence and ranks \(A = true\) as the top fault-relevant predicate.
This predicate readily leads the developer to the fault location. Meanwhile, we note that because
neither \(A = true\) nor \(A = false\) is an invariant in correct or incorrect executions, invariant-based
methods cannot detect that A is a suspicious predicate. Liblit05 does not regard A as suspicious
either because it does not model the predicate evaluation within each execution (see Section 3.3.7
for details).

The above example illustrates a simple but representative case where a probabilistic treatment
of predicates captures detailed information about predicate evaluations. In the next section, we
describe the statistical model and the ranking algorithm that implement this intuition.
3.3 Predicate Ranking Models

3.3.1 Problem Settings

Let $T = \{t_1, t_2, \cdots, t_n\}$ be a test suite for program $P$. Each test case $t_i = (d_i, o_i)$ ($1 \leq i \leq n$) has an input $d_i$ and the expected output $o_i$. The execution of $P$ on each test case $t_i$ gives the output $o'_i = P(d_i)$. We say $P$ passes the test case $t_i$ (i.e., $t_i$ is a passing case) if and only if $o'_i$ is identical to $o_i$; otherwise, $P$ fails on $t_i$ (i.e., $t_i$ is a failing case). In this way, the test suite $T$ is partitioned into two disjoint subsets $T_p$ and $T_f$, corresponding to the passing and failing cases respectively,

$$T_p = \{t_i | o'_i = P(d_i) \text{ and } o'_i = o_i\},$$

$$T_f = \{t_i | o'_i = P(d_i) \text{ and } o'_i \neq o_i\}.$$

Since program $P$ passes test case $t_i$ if and only if $P$ executes correctly, we use “correct” and “passing”, “incorrect” and “failing” interchangeably in the following discussion.

Given a faulty program $P$ together with a test suite $T = T_p \cup T_f$, our task is to localize the suspicious fault region by contrasting $P$’s runtime behaviors on $T_p$ and $T_f$.

3.3.2 Probabilistic Treatment of Predicates

In general, a program predicate is a proposition about any program property, such as “$\text{idx} < \text{LENGTH}$”, “!empty(list)” and “$\text{foo}() > 0$”. As any instrumentation site can be touched more than once due to program control flows, a predicate $P$ can be evaluated multiple times in one execution, and each evaluation produces either true or false. In order to model this within-execution behavior of $P$, we develop the concept of evaluation bias, which estimates the probability of the predicate $P$ being evaluated as true.

**Definition 1 (Evaluation Bias)** Let $n_t$ be the number of times that predicate $P$ evaluates to true, and $n_f$ the number of times it evaluates to false, in one execution. $\pi(P) = \frac{n_t}{n_t + n_f}$ is the observed evaluation bias of predicate $P$ in this particular execution.

Intuitively, $\pi(P)$ estimates the probability that $P$ takes the value true in each evaluation. If the instrumentation site of $P$ is touched at least once (i.e., $n_t + n_f \neq 0$), $\pi(P)$ varies in the range
of $[0, 1]$: $\pi(P)$ is equal to 1 if $P$ always holds, to 0 if it never holds, and in between for all other sets of outcomes. If the predicate is never evaluated, $\pi(P)$ has a singularity $0/0$. In this case, since we have no evidence to favor either true or false, we set $\pi(P)$ to 0.5 for fairness. Finally, if a predicate is never evaluated in any failing runs, it has nothing to do with program failures and is hence eliminated from the predicate ranking.

### 3.3.3 Methodology Overview

In this section, we formulate the main idea of our method, and then develop its details in the next section. Following the convention in statistics, we use uppercase letters for random variables and lowercase letters for their realizations. Moreover, $f(X|\theta)$ is a general notation of the probability model for the random variable $X$ that is indexed by the parameter $\theta$.

Let the entire test case space be $T$, which conceptually contains all the possible inputs and expected outputs. According to the correctness of $P$ on the test cases in $T$, $T$ can be partitioned into two disjoint sets $T_p$ and $T_f$ for passing and failing cases. Therefore, the available test suite $T$ and its partitions $T_p$ and $T_f$ can be treated as a random sample from $T$, $T_p$ and $T_f$ respectively. Let $X$ be the random variable for the evaluation bias of predicate $P$. We then use $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$ to denote the statistical model for the evaluation bias of $P$ in $T_p$ and $T_f$ respectively. Therefore, the evaluation bias from running a test case $t$ can be treated as an observation from $f_P(X|\theta)$, where $\theta$ is either $\theta_p$ or $\theta_f$ depending on whether $t$ is passing or failing. Given the statistical models for both passing and failing runs, we then define the fault relevance of $P$ as follows.

**Definition 2 (Fault Relevance)** A predicate $P$ is relevant to the hidden fault if its underlying model $f_P(X|\theta_f)$ diverges from $f_P(X|\theta_p)$, where $X$ is the random variable for the evaluation bias of $P$.

The above definition relates $f_P(X|\theta)$, the statistical model for $P$’s evaluation bias, with the hidden fault. Naturally, the larger the difference between $f_P(X|\theta_f)$ and $f_P(X|\theta_p)$, the more relevant $P$ is to the fault. Let $L(P)$ be an arbitrary similarity function,

$$L(P) = \text{Sim}(f(X|\theta_p), f(X|\theta_f)).$$  (3.1)
The ranking score $s(P)$ can be defined as $g(L(P))$. Because we are only interested in the relative ranking of predicates, $g(x)$ can be any monotonically decreasing function. We here choose $g(x) = -\log(x)$ because $\log(x)$ effectively measures the comparative magnitude when $x$’s are closed to 0 (certainly, $x$ must be positive). Therefore, the fault relevance score $s(P)$ is defined as

$$s(P) = -\log(L(P)).$$  \hspace{1cm} (3.2)$$

Using this fault relevance score, we can rank all the instrumented predicates, and the top ranked ones are regarded more likely to be fault-relevant. Therefore, the fault localization problem boils down to the setting of the similarity function, which, in turn, consists of two sub-problems: (1) What is a suitable similarity function $L(P)$, and (2) how is $L(P)$ computed when the closed form of $f_P(X|\theta)$ is unknown? In the following subsections, we examine the two problems in detail.

### 3.3.4 Predicate Ranking

The lack of prior knowledge about $f_P(X|\theta)$ constitutes one of the major obstacles in calculating the similarity (or difference equivalently) between $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$. If the closed form of $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$ were given, measures used in information theory [26], such as the relative entropy, would immediately apply. Meanwhile, we are not authorized to impose model assumptions, like normality, on $f_P(X)$ because improper assumptions can lead to misleading inferences. Therefore, given the above difficulties in directly measuring the model difference, in this article we propose an indirect approach that measures the difference between $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$ without any model assumption.

Aiming at the model difference, we first propose the null hypothesis that $\mathcal{H}_0$: $f_P(X|\theta_p) = f_P(X|\theta_f)$, i.e., no difference exists. Let $X = (X_1, X_2, \cdots, X_m)$ be a random sample from $f_P(X|\theta_f)$ (i.e., observed evaluation bias from $m$ failing cases), we derive a statistic $Y$, which, under the null hypothesis $\mathcal{H}_0$, conforms to a known distribution. If the realized statistic $Y(X)$ corresponds to an event that has a small likelihood of happening, the null hypothesis $\mathcal{H}_0$ is likely invalid, which suggests that a nontrivial difference exists between $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$.

We choose to characterize $f_P(X|\theta)$ through its population mean $\mu$ and variance $\sigma^2$, so that the
null hypothesis $\mathcal{H}_0$ is
\[
\mu_p = \mu_f \text{ and } \sigma_p^2 = \sigma_f^2. \tag{3.3}
\]

Let $X = (X_1, X_2, \cdots, X_m)$ be an independent and identically distributed (i.i.d.) random sample from $f_p(X|\theta_f)$. Under the null hypothesis, we have $E(X_i) = \mu_f = \mu_p$ and $Var(X_i) = \sigma_f^2 = \sigma_p^2$. Because $X_i \in [0, 1]$, both $E(X_i)$ and $Var(X_i)$ are finite. According to the Central Limit Theorem [20], the following statistic
\[
Y = \frac{\sum_{i=1}^m X_i}{m}, \tag{3.4}
\]
asymptotically conforms to $N(\mu_p, \frac{\sigma_p^2}{m})$.

Let $f(Y|\theta_p)$ be the probability density function of the normal distribution $N(\mu_p, \frac{\sigma_p^2}{m})$. Then the likelihood $L(\theta_p|Y)$ of $\theta_p$ given the observed $Y$ is
\[
L(\theta_p|Y) = f(Y|\theta_p). \tag{3.5}
\]

A smaller likelihood implies that $\mathcal{H}_0$ is less likely to hold, which, in turn, indicates a larger difference between $f_p(X|\theta_p)$ and $f_p(X|\theta_f)$. Therefore, we can reasonably instantiate the similarity function in Eq. (3.1) with the likelihood function
\[
\mathbf{L}(P) = L(\theta_p|Y). \tag{3.6}
\]

According to the property of normal distribution, the normalized statistic
\[
Z = \frac{Y - \mu_p}{\sigma_p/\sqrt{m}} \tag{3.7}
\]
asymptotically conforms to the standard normal distribution $N(0,1)$, and
\[
f(Y|\theta_p) = \frac{\sqrt{m}}{\sigma_p} \varphi(Z), \tag{3.8}
\]
where $\varphi(Z)$ is the probability density function of $N(0,1)$. 24
Combining Eqs. (3.2), (3.6), (3.5), and (3.8), we finally get the ranking score for predicate $P$ as

$$ s(P) = -\log(L(P)) = \log\left(\frac{\sigma_p}{\sqrt{m\varphi(Z)}}\right). $$

(3.9)

### 3.3.5 Discussions on Score Computation

First, in order to calculate $s(P)$ using Eq. (3.9), we need to estimate the population mean $\mu_p$ and the standard error $\sigma_p$ of $f_P(X|\theta_p)$. Let $X' = (X'_1, X'_2, \cdots, X'_n)$ be a random sample from $f_P(X|\theta_p)$ (which corresponds to the observed evaluation bias from the $n$ passing runs), then $\mu_p$ and $\sigma_p$ can be estimated as

$$ \mu_p = \overline{X'} = \frac{\sum_{i=1}^{n} X'_i}{n} $$

(3.10)

and

$$ \sigma_p = S_{X'} = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (X'_i - \overline{X'})^2}. $$

(3.11)

Secondly, because the $\sqrt{m}$ in Eq. (3.9) does not affect the relative order between predicates, it can be safely dropped in practice. However, as simple algebra would reveal, the $m$ in Eq. (3.4) and the $\sqrt{m}$ in Eq. (3.7) cannot be discarded, because they properly scale the statistics for standard normality as required by the Central Limit Theorem.

Finally, we note that although the derivation of Eq. (3.9) is based on the asymptotic behavior, i.e., when $m \to +\infty$, statistical inference suggests that the asymptotic result is still valid even when the sample size is nowhere near infinity [20]. In our fault localization scenario, it is true that we cannot have an infinite number of failing runs. But as shown in experiments, Eq. (3.9) still works well in ranking abnormal predicates even when only a small number of failing runs is available.

We now use a concrete example to conclude the discussion in this subsection. It illustrates how the fault relevance score of the predicate $P = (A = \text{true})$ is calculated for the program in Fig. 3.1.

First, by running the 130 failing and 5412 passing cases (i.e., $m = 130$ and $n = 5412$) on the instrumented program, the numbers of true and false evaluations are recorded at runtime for each execution. Then the evaluation bias of $P$ in each execution is calculated based on Definition 1. Next, the statistic $Y = 0.9024$ is directly obtained from the evaluation biases in failing cases according to Eq. 3.4. Similarly, from passing cases, we get $\mu_p = 0.2952$ and $\sigma_p = 0.2827$ according
to Eq. 3.10 and Eq. 3.11, respectively. Plugging the calculated $Y$, $\mu_p$, $\sigma_p$ and $m = 130$ into Eq. 3.7, we get $Z = 24.4894$. Finally, from Eq. 3.9, the fault relevance score for predicate $P$ is 296.2.

Besides illustrating how $s(P)$ is calculated, this example also shows the role played by the log operator in Eq. 3.9. Although the log operator does not influence the ranking of predicates, it helps scale down the calculated score, which might otherwise overflow in numeric computation.

3.3.6 Generalizing Invariants

In this section, we demonstrate how the probabilistic treatment of predicate evaluations encompasses program invariants [34] as a special case. Moreover, we also prove that the fault relevance score function of Eq. (3.9) readily identifies both invariant violations and conformations.

Without loss of generality, a predicate $P$ is a program invariant on a test suite $C$ if and only if it always evaluates true during the execution of $C$. In practice, $C$ is usually chosen to be a set of passing cases so that the summarized invariants characterize the correct behavior of the subject program [34]. During the faulty executions, these invariants are either conformed (i.e., still evaluate true) or violated (i.e., evaluate false at least once), and those violated invariants are regarded as hints for debugging. In some special cases, the test suite $C$ is chosen to be a time interval during which the execution is believed to be correct. One typical example is that for software that runs for a long time, such as web servers, the execution is likely to be correct at the beginning of the run [39].

According to Definition 1, the evaluation bias of an invariant is always 1. Taking the set of passing cases $T_p$ as $C$, we know that, if the predicate $P$ is an invariant, $\mu_p = 1$ and $\sigma_p = 0$. Moreover, the following theorem proves that the score function of Eq. (3.9) naturally identifies both invariant violations and conformations.

**Theorem 1** Let $P$ be any invariant summarized from a set of correct executions $T_p$, $s(P) = +\infty$ if $P$ is violated in at least one faulty execution; and $s(P) = -\infty$ if $P$ is conformed in all faulty executions.

Theorem 1 indicates that if a fault can be caught by invariant violations as implemented in the DIDUCE [39] project, SOBER can also detect it because the fault relevance score for a violated
Let \( x = (x_1, x_2, \cdots, x_m) \) be a realized random sample, which corresponds to the observed evaluation biases from the \( m \) failing runs. Once \( P \) is violated in at least one execution, \( \sum_{i=1}^{m} x_i \neq m \). It then follows from Eq. (3.7) that

\[
z = \frac{c}{\sigma_p} \quad \text{where} \quad c = \frac{\sum_{i=1}^{m} x_i - m\mu_p}{\sqrt{m}} \neq 0,
\]

then

\[
\lim_{\sigma_p \to 0} \frac{\sigma_p}{\sqrt{m}\varphi(z)} = \sqrt{\frac{2\pi}{m}} \lim_{\sigma_p \to 0} \frac{\sigma_p}{e^{-\frac{1}{2} \left( \frac{c^2}{\sigma_p^2} \right)}} = \sqrt{\frac{2\pi}{m}} \lim_{t \to \infty} \frac{e^{-\frac{1}{2} \left( \frac{c^2}{t} \right)}}{t}
\]

\[
= c^2 \sqrt{\frac{2\pi}{m}} \lim_{t \to \infty} te^{\frac{c^2 t}{2}} = +\infty.
\]

Thus Eq. (3.9) gives \( s(P) = +\infty \). This means that SOBER treats violated invariants as the most abnormal predicates and ranks them at the top.

On the other hand, if the invariant \( P \) is not violated in any failing run, we have

\[
\lim_{\sigma_p \to 0} z = \lim_{\sigma_p \to 0} \frac{\sum_{i=1}^{m} x_i - m\mu_p}{\sqrt{m}\sigma_p} = \lim_{\sigma_p \to 0} \frac{0}{\sqrt{m}\sigma_p} = 0,
\]

and therefore,

\[
\lim_{\sigma_p \to 0} \frac{\sigma_p}{\sqrt{m}\varphi(z)} = \lim_{\sigma_p \to 0} \frac{\sigma_p}{\sqrt{m}\varphi(0)} = 0,
\]

which immediately leads to \( s(P) = -\infty \). This suggests that conformed invariants are regarded the least abnormal, and are ranked at the bottom by our method.
invariant is $+\infty$. Meanwhile, for conformed invariants, SOBER simply discards them due to the $-\infty$ score. Previous research suggests that invariant violations themselves can only locate a number of faults in the Siemens suite [79]. As will be shown shortly, our method SOBER, being a superset of invariant-based methods, actually achieves the best fault localization results on the Siemens suite.

### 3.3.7 Differences between SOBER and Liblit05

Because both LIBLIT05 and SOBER are based on a statistical analysis of predicate evaluations, we now illustrate the differences in this subsection. We first outline the basic idea of LIBLIT05.

In principle, LIBLIT05 contrasts the probability that one execution crashes if the predicate $P$ is ever observed true, and that if $P$ is observed (either true or false) in the execution. Specifically, the authors define

\[
\text{Context}(P) = P_r(\text{Crash} | P \text{ observed}),
\]
\[
\text{Failure}(P) = P_r(\text{Crash} | P \text{ observed true}),
\]

and take the probability difference

\[
\text{Increase}(P) = \text{Failure}(P) - \text{Context}(P),
\]

as one of the two key components of $P$’s fault relevance score. The other component is the number of failing runs where $P$ is ever observed as true. A harmonic average is then taken to combine these two components.

A detailed examination reveals fundamental differences between LIBLIT05 and SOBER. First, from the methodological point of view, LIBLIT05 estimates how much more likely an execution crashes if the predicate $P$ is observed as true in comparison with if $P$ is observed as either true or false. This indicates that LIBLIT05 places a greater value on predicates whose true evaluation correlates with program crashes. SOBER, on the other hand, models the evaluation distribution of the predicate $P$ in passing (i.e., $f_P(X|\theta_p)$) and failing (i.e., $f_F(X|\theta_f)$) runs respectively, and regards predicates with large differences between $f_P(X|\theta_f)$ and $f_F(X|\theta_p)$ as fault-relevant. Therefore, SOBER and LIBLIT05 actually follow two fundamentally different approaches, although both of
Figure 3.4: Methodology Difference between Liblit05 and SOBER

them rank predicates statistically. Secondly, SOBER explores the multiple evaluations of predicates within one execution while Liblit05 overlooks this information. For instance, if a predicate $P$ evaluates as true at least once in each execution, and has different likelihood to be true in passing and failing runs, Liblit05 simply overlooks $P$ while SOBER can readily capture the evaluation divergence.

Figure 3.4 provides a visual illustration of the methodology difference between Liblit05 and SOBER. We use the circle to denote the whole set of tested cases, and the top part represents failing cases (denoted by crosses, here 6), and the bottom part for the passing cases (denoted by circles, here 10). Now let us discuss how the fault-relevance score is computed each algorithm for a given predicate $P$. The algorithm Liblit05 first computes Context($P$) by checking in how many cases out of the total 16 the predicate $P$ is ever evaluated, and this set is denoted by the outer shaded region in Figure 3.4(a). With this set located, we get Context($P$) = 4/10 = 2/5. Then, Liblit05 further checks within the located set of cases, in how many of them the predicate $P$ is ever evaluated as true, and this set is denoted by the inner shaded region in Figure 3.4(a). Within this region, we can get the ratio of failing cases, which is exactly the Failure($P$), i.e., Failure($P$) = 3/7. Finally, the fault-relevance score of predicate $P$ is the difference $\frac{1}{35}$.

On the other hand, Figure 3.4(b) demonstrates that SOBER actually implements a totally different methodology from Liblit05. Specifically, SOBER calculates the evaluation bias for the predicate $P$ from each execution according to Definition 1, and treats the evaluation biases from passing and failing executions as random samples from the models $f(X|\theta_p)$ and $f(X|\theta_f)$, respectively. Finally, the Equation 3.1 is used to calculate the fault relevance score of predicate $P$ for SOBER. From
this visual comparison, one can notice that the effectiveness of LIBLIT05 relies on significance of
the difference between ratios of failing executions over different subsets. As an extreme case, if a
predicate is ever evaluated true in most passing and failing executions, it has little discriminant
power for fault localization even if its evaluation patterns are different between passing and failing
executions. Although this seems to be an extreme case, it does happen frequently because most
predicates sit inside certain loops and will likely get evaluated as true at least once. Let us go
back to the example in Fig. 3.1 to clarify such situations.

The faulty statement (Line 7) is executed in almost every execution. Within each run, it
evaluates multiple times as either true or false. In this case, LIBLIT05 has little discrimination
power. Specifically, for the predicate $P : \neg(m \geq 0) \Rightarrow true$, $Increase(P) = 0.0104$ and the
$Increase$ value for predicate $P' : \neg(m \geq 0) \Rightarrow false$ is $-0.0245$. According to [59], neither $P$
or $P'$ is ranked on top since they are either negative or too small. Thus, LIBLIT05 fails to identify
the defect point. As our experiments show, SOBER successfully ranks $P$ as the most suspicious
predicate. Intuitively, this is because the evaluation bias in failing runs (0.9024) significantly
diverges from that in passing ones (0.2952).

Finally, we note that although the above discussion reveals some advantages of SOBER over
LIBLIT05, we cannot claim SOBER is consistently better than LIBLIT05 for any cases. In fact,
there are certain cases where LIBLIT05 does outperform SOBER. For example, we notice that when
the percentage of failing executions is higher and the fault is not inside certain loops, LIBLIT05 tends
to rank the most fault-relevant predicates higher than SOBER. But these are simply conjectures.
Because the mechanism of LIBLIT05 and SOBER is fundamentally different, and there are many
other factors besides the debugging mechanism (e.g., the quality of the used test suite) that can also
affect the debugging result, we cannot theoretically prove under what circumstances one debugging
algorithm is better than the other, although such result, if obtained, is really invaluable. Therefore,
we still leave the comparison to empirical evaluations on a set of subject programs, as will be
presented in the following.
Table 3.1: Characteristics of Subject Programs

<table>
<thead>
<tr>
<th>Faulty Ver.</th>
<th>LOC</th>
<th>Test #</th>
<th>Median of Failing</th>
<th>Median of Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>7</td>
<td>539</td>
<td>4130</td>
<td>38</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>489</td>
<td>4115</td>
<td>223</td>
</tr>
<tr>
<td>replace</td>
<td>32</td>
<td>507</td>
<td>5542</td>
<td>83</td>
</tr>
<tr>
<td>schedule</td>
<td>9</td>
<td>397</td>
<td>2650</td>
<td>31</td>
</tr>
<tr>
<td>schedule2</td>
<td>10</td>
<td>299</td>
<td>2710</td>
<td>32</td>
</tr>
<tr>
<td>tcas</td>
<td>41</td>
<td>174</td>
<td>1608</td>
<td>23</td>
</tr>
<tr>
<td>tot_info</td>
<td>23</td>
<td>398</td>
<td>1052</td>
<td>71</td>
</tr>
</tbody>
</table>

3.4 Empirical Comparison with Existing Techniques

In this section, we empirically evaluate the effectiveness of Sober in fault localization. We compare Sober with seven existing fault localization algorithms under the same setting as previous studies. Section 5.4.1 first describes the experimental setup, which includes the used subject programs, the metric for localization quality, and the implementation details. We briefly explain the seven fault localization algorithms in Section 3.4.2. Detailed comparison results are presented in Sections 3.4.3 and 3.4.4. Finally, Section 3.4.5 compares these algorithms from different perspectives other than the localization accuracy.

3.4.1 Experimental Setup

In this study, we use the Siemens suite as our subject programs. The Siemens suite was originally prepared by Siemens Corp. Research in a study of test adequacy criteria [44]. It contains 132 faulty versions of seven subject programs, where each faulty version contains one and only one manually injected fault. Table 3.1 lists the characteristics of the seven subject programs. The medians of the failing and passing cases are taken over all the faulty versions of each subject program. Readers interested in more details about the Siemens suite are referred to [44] [82].

Previously, many researchers investigating fault localization have reported their results on the Siemens suite [41] [79] [81] [23]. Because no failures are observed for the 32nd version of the replace program and the 10th version of the schedule2 program on the supplied test suites, these two versions are excluded in previous studies [81, 23, 64], as well as this one.
In order to objectively quantify the localization accuracy, an evaluation framework based on program static dependencies is adopted in this study. This measure was originally proposed by Renieris et al. [81], and was later adopted by Cleve et al. in reporting the quality of CTr [23]. We briefly summarize this measure as follows.

1. Given a (faulty) program \( P \), its program dependence graph (PDG) is written as \( G \), where each statement is a node and there is an edge between two nodes if two statements have data and/or control dependencies.

2. The nodes corresponding to faulty statements are marked as defect nodes. The set of defect nodes is written as \( V_{\text{defect}} \).

3. Given a fault localization report \( R \), which is a set of suspicious statements, their corresponding nodes are called blamed nodes. The set of blamed nodes is written as \( V_{\text{blamed}} \).

4. A developer can start from \( V_{\text{blamed}} \) and perform a breadth-first search until he reaches one of the defect nodes. The set of statements covered by the breadth-first search is written as \( V_{\text{examined}} \).

5. The \( T \)-score, defined as follows, measures the percentage of code that has been examined in order to reach the fault,

\[
T = \frac{|V_{\text{examined}}|}{|V|} \times 100\%,
\]

where \(|V|\) is the size of the program dependence graph \( G \). In [81, 23], the authors used \( 1 - T \) as an equivalent measure.

The \( T \)-score estimates the percentage of code a developer needs to examine (along the static dependencies) before the fault location is found, when a fault localization report is provided. A high quality fault localization is expected to be a small set of statements that are close to (or contain) the fault location. The above definition of \( T \)-score is immediately applicable to localizations that consist of a set of “blamed” statements. For algorithms that generate a ranked list of all predicates, like LIBLIT05 and SOBER, the corresponding statements of the top-\( k \) predicates are taken as a fault localization report. The optimal \( k \) is the one that minimizes the average examined code over a set of faults under study, i.e.,

\[
k_{\text{opt}} = \arg\min_k E[T_k].
\]
where $E[T_k]$ is the average $T$-Score for the given set of faults for any fixed $k$.

As the above defined $T$-score is calculated based on PDGs, we call it PDG-based. Recently, another kind of $T$-score was used by Jones et al. in reporting the localization results of TARANTULA [47]. The TARANTULA tool produces a ranking of all executable statements, and the authors calculate the $T$-score directly from the ranking. Instead of taking the top-$k$ statements, and calculating the $T$-score based on PDGs, the authors examine whether the faulty statements are ranked high. Specifically, a developer is assumed to examine statement by statement from the top of the ranking until a faulty statement is touched. The percentage of examined statements by then is taken as the $T$-score. We call the $T$-score calculated in this way ranking-based. Apparently, the ranking-based $T$-score assumes a different code examination strategy than that assumed by the PDG-based, i.e., along the ranking rather than along the dependencies. Intuitively, the PDG-based approach is closer to practice. Moreover, the ranking-based $T$-score is not as generally applicable as the PDG-based, because it requires a ranking of all statements. For example, none of the discussed algorithms in Section 3.4.2 except TARANTULA, can be evaluated using the ranking-based approach, but TARANTULA can be evaluated by the PDG-based $T$-score by taking the top-$k$ statements as a fault localization report.

Finally, we explain why precision/recall (or equivalently the false positive and false negative rate) is not used for evaluation, now that the debugging result from SOBER is a predicate ranking. In information retrieval, given a query $q$, the retrieval result is a ranking of documents with top ranked ones as being more likely to be relevant to $q$, and the precision/recall is a standard metric for evaluation. Although the debugging result from both SOBER and LIBLIT05 is also a ranking (of predicates), precision/recall is not applicable because we do not have the ground truth that which predicates are fault relevant and which are not. In the setting of information retrieval, the precision/recall is used for evaluation because we know whether a document is relevant or not. But for the debugging result, we cannot determine whether a predicate is fault relevant; instead, what we can determine is merely how far a predicate is from the real fault location. For the same reason, we cannot use the false positive and false negative rates to measure the quality of predicate ranking. In a previous study of debugging system misconfiguration problems [91], where the debugging result is a ranking of suspicious registry entries on Windows machines, the authors
evaluate the debugging quality by reporting the rank of the root cause registry entry (they focus on misconfiguration cases that are each caused by one registry entry). We cannot evaluate our debugging result in this way because there is not necessarily a predicate on the root cause location in the source code. Therefore, we adopt the PDG-based T-score, which is applicable to all the debugging algorithms compared in this study.

We compare SOBER with seven existing fault localization algorithms (described in the next subsection). Among them, we implemented LIBLIT05 in Matlab, and validated the correctness of the implementation with the original authors. For the other six algorithms, the localization result on the Siemens suite is taken directly from their corresponding publications.

We instrumented the subject programs with two kinds of predicates: branches and function returns, which are described in detail in [58, 59]. In particular, we treat each branch conditional as one inseparable instrumentation unit, and do not consider each subclause separately. For better fault localization, one may be tempted to introduce more predicates. But the introduction of more predicates is a double-edged sword. On the positive side, an expanded set of predicates is more likely to cover the faulty code; but the superfluous predicates brought in can nontrivially complicate the predicate ranking. So far, no agreement has been reached on what are the “golden predicates”. At runtime, the evaluation of predicates is collected without sampling for both LIBLIT05 and SOBER.

All experiments in this section were carried out on a 3.2GHz Intel Pentium-4 PC with 1GB physical memory, running Fedora Core 2. In calculating the T-scores, we used CODESURFER 1.9 with patch 3 to generate the program dependence graphs. Because PDGs generated by CODESURFER may vary with different build options, the factory default (by enabling the factory-default switch) is used to allow reproducible results in the future. Moreover, the Matlab source code of SOBER and the instrumented Siemens suite are available online at http://www.ews.uiuc.edu/~chaoliu/sober.htm.

3.4.2 Compared Fault Localization Algorithms

We now briefly explain the seven fault localization algorithms we compare with SOBER. As LIBLIT05 is already discussed in Section 3.3.7, we only describe the other six algorithms below.
- **Set-union**: This algorithm is based on the program spectra difference between a failing case $f$ and a set of passing cases $P$. Specifically, let $S(t)$ be the program spectra of running the test case $t$, then the set difference between $S(f)$ and the union spectra of cases in $P$ is taken as the fault localization report $R$, i.e., $R = S(f) - \cup_{p_i \in P} S(p_i)$. This algorithm is described in [81], and we denote it by UNION for brevity.

- **Set-intersect**: A complementary algorithm to UNION is also described in [81]. It is based on the set difference between the spectra of the failing case and the intersection spectra of passing cases, namely, the localization report $R = \cap_{p_i \in P} S(p_i) - S(f)$. We denote this algorithm by INTERSECT.

- **Nearest Neighbor**: The nearest neighbor approach, proposed by Renieris and Reiss in [81], contrasts the failing case to the passing case that most “resembles” the failing case. Namely, the localization report $R = S(f) - S(p)$, where $p$ is the nearest passing case to $f$ as measured under certain distance metrics. The authors studied two distance metrics, and found that the nearest neighbor search based on the Ulam’s distance renders better fault localization. This algorithm is denoted as NN/PERM by the original authors.

- **Cause Transition**: The Cause Transition algorithm [23], denoted as CT, is an enhanced variant of Delta Debugging [95]. Delta Debugging contrasts the memory graph [102] of one failing execution $e_f$ against that of one passing execution $e_p$. By carefully manipulating the two memory graphs, Delta Debugging systematically narrows the difference between $e_f$ and $e_p$ down to a small set of suspicious variables. CT enhances Delta Debugging by exploiting the notion of cause transition: “moments where new relevant variables begin being failure causes” [23]. Therefore, CT essentially implements the concept of “search in time”, in addition to the original “search in space” used in Delta Debugging.

- **Tarantula**: The TARANTULA tool was originally presented to visualize the test information for each statement in a subject program, and it was shown to be useful for fault localization [46]. In a recent study [47], the authors took $(1 - \text{hue}(s))$ as the fault relevance score for the statement $s$, where $\text{hue}(s)$ is the hue component of each statement in visualization [46]. With the fault relevance score calculated for each statement, TARANTULA produces a ranking of all executable
statements. Developers are expected to examine the ranking from the top down to locate the fault.

- **Failure-Inducing Chops**: Gupta et al. recently propose a fault localization algorithm that integrates delta debugging and dynamic slicing [38]. First, a minimal failure-inducing input $f'$ is derived from the given failing case $f$ using the algorithms of Zeller and Hildebrandt [96]. Then a forward dynamic slice $FS$, and a backward slice $BS$ are calculated from $f'$ and the erroneous output, respectively. Finally, the intersection of $FS$ and $BS$, i.e., the chop, is taken as the fault localization report, namely, $R = FS \cap BS$. We denote this algorithm by SLICECHOP.

In previous studies, comparisons of some of the above algorithms are reported. Specifically, Renieris and Reiss found that NN/PERM outperformed both UNION and INTERSECT [81], whereas Cleve and Zeller later reported that a better result than NN/PERM was achieved by Ct [23]. These reported results are all based on the PDG-based $T$-score. As Ct achieves the best localization result as measured with the PDG-based $T$-score, we compare SOBER with Ct and LIBLIT05 in Section 3.4.3 using the same measure. Because TARANTULA and SLICECHOP results are not reported with the PDG-based $T$-score, we compare SOBER with them separately in Section 3.4.4.

### 3.4.3 Comparison with Liblit05 and CT

In this section, we compare SOBER with Ct and Liblit05. We subject both Liblit05 and SOBER to the 130 faults in the Siemens suite and measure their localization quality using the PDG-based...
T-Score (Eq. (3.15)). The result of Ct is directly cited from [23].

Fig. 3.5(a) depicts the number of faults that can be located when a certain percentage of code is examined by a developer. The x-axis is labelled with T-Score. For Liblit05 and Sober, we choose the top-5 predicates to form the set of blamed nodes. Because localization that still requires developers to examine more than 20% of the code is generally useless, we only treat [0, 20] as the meaningful T-Score range. Under these circumstances, Sober is apparently better than Liblit05 while both of them are consistently superior to Ct.

For practical use, it is instructive to know how many (or what percentage of) faults can be identified when no more than α% of the code is examined. We therefore plot the cumulative comparison in Fig. 3.5(b). It clearly suggests that both Sober and Liblit05 are much better than Ct, and that Sober outperforms Liblit05 consistently. Although Liblit05 catches up when the T-Score is 60% and higher, we regard this advantage as irrelevant because it barely makes sense for a fault locator to require a developer to examine more than 60% of the code.

Fig. 3.5(b) shows that for the 130 faults in the Siemens suite, when a developer examines at most 1% of the code, Ct catches 4.65% of the faults while Liblit05 and Sober capture 7.69% and 8.46%, respectively. Moreover, when 10% code examination is acceptable, Ct and Liblit05 identify 34 (26.36%) and 52 (40.00%) out of the 130 faults. Sober is the best of the three, locating
68 (52.31%) out of the 130 faults, which is 16 faults more than the state-of-the-art approach Liblit05. If the developer is patient enough to examine 20% of the code, 73.85% of the faults (i.e., 96 out of 130) can be located by SOBER.

![Figure 3.7: Quality of SOBER w.r.t. top-k Values](image)

We also vary the parameter $k$ in calculating the $T$-score for both Liblit05 and SOBER. The quality comparison is plotted in Fig. 3.6 for $k$ varying from 1 through 8. The comparison is confined within the [0, 20] $T$-score range. Since detailed results about $C_T$ is not available in [23], $C_T$ is still depicted only at the 1, 10, and 20 ticks. Fig. 3.6 shows that Liblit05 is the best when $k$ is equal to 1 and 2. When $k = 3$, SOBER catches up, and consistently outperforms Liblit05 afterwards. Because developers are always interested in locating faults with minimal code checking, it is desirable to select the optimal $k$ that maximizes the localization quality. We found that both Liblit05 and SOBER achieve their best quality when $k$ is equal to 5. In addition, Fig. 3.7 plots the quality of SOBER with various $k$-values. It clearly indicates that SOBER locates the largest number of faults when $k$ is equal to 5. Therefore, the setting of $k = 5$ in Fig. 3.5 is justified. Finally, Fig. 3.7 also suggests that too few predicates (e.g., $k = 1$) may not convey enough information for fault localization, while too many predicates (e.g., $k = 9$) are in themselves a burden for developers to examine, and thus neither of them leads to best results. In practice, we can either return the entire predicate ranking to developers so that they can check predicates from the top down, or we can consult developers’ opinions on the appropriate $k$ they feel comfortable, and only return the top-$k$ predicates.

Besides being accurate in fault localization, SOBER is also computationally efficient. Suppose
we have \( n \) correct and \( m \) incorrect executions, then the time complexity of scoring each predicate is \( O(n + m) \). If there are in total \( k \) predicates instrumented, the entire time complexity of \text{SOBER} is \( O((n + m) \cdot k + k \cdot \log(k)) \). Similarly, \text{LIBLIT05} also needs \( O(n + m) \) to score each predicate, and its time complexity is also \( O((n + m) \cdot k + k \cdot \log(k)) \). We experimented with the 31 faulty versions of the \text{replace} program, the average time for un-optimized \text{LIBLIT05} and \text{SOBER} to analyze each version is 11.7775 and 11.3844 seconds respectively. This is much faster than \text{CT}, as reported in [23].

### 3.4.4 Comparison with Tarantula and SliceChop

We now compare \text{SOBER} with \text{TARANTULA} and \text{SLICECHOP}. Recently, Jones et al. reported the result of \text{TARANTULA} on the Siemens suite with the ranking-based \( T \)-score, and compared it with previous PDG-based \( T \)-scores of \text{CT}, \text{NN/PERM}, \text{INTERSECT} and \text{UNION}. As it is unclear to what extent these two kinds of \( T \)-score agree with each other, we assume they are equivalent, as did by Jones et al. in [47]. More investigation, however, is needed to clarify this issue in the future. Moreover, because the authors failed to compare \text{TARANTULA} with statistical debugging in [47], this study fills the gap.

We differ from previous comparisons in choosing to compare algorithms in terms of the absolute number of faulty versions on which an algorithm renders a \( T \)-score of no more than \( \alpha \)%. Previously, different subsets of the Siemens suite were used by different authors, and the percentages based on the different subsets are put together for comparison [81, 23, 47, 38]. Specifically, the reported percentages for \text{UNION}, \text{INTERSECT} and \text{NN/PERM} are based on 109 faulty versions, and the percentage for \text{CT} is based on 129 versions. In the previous subsection, the percentages for \text{LIBLIT05} and \text{SOBER} are calculated on the whole-set 130 versions. In a recent study of \text{TARANTULA} and \text{SLICECHOP} [47, 38], 122 and 38 faulty versions are used by the original authors.

Therefore, based on the reported percentage and the chosen subset of faulty versions, we recover how many faults are located by each algorithm with a \( T \)-score no more than \( \alpha \)%, and Fig. 3.8 shows the effectiveness comparison in terms of the absolute number of faulty versions. Because the study of \text{SLICECHOP} excluded 91 faulty versions, for fairness it is not plotted in Fig. 3.8. Instead, we compare \text{SOBER} with \text{SLICECHOP} separately later.
Fig. 3.8 clearly shows that the effectiveness of the seven algorithms is at three different levels. The algorithms \texttt{Union} and \texttt{Intersect} are the least effective, \texttt{NN/Perm} and \texttt{CT} are in the middle, with \texttt{CT} being better than \texttt{NN/Perm}. The other three algorithms: \texttt{Liblit05}, \texttt{Tarantula} and \texttt{Sober}, apparently have the best result on the Siemens suite.

We now compare \texttt{Tarantula} with \texttt{Sober} in detail. They both locate 68 faults when the $T$-score is no more than 10\%. When the $T$-score is less than 1\%, \texttt{Tarantula} and \texttt{Sober} locate 17 and 11 faults, respectively. On the other hand, with the $T$-score no more than 20\%, \texttt{Sober} can help locate 96 out of the 130 faults whereas \texttt{Tarantula} helps locate 75. However since the comparison is based on the assumption of the equivalence between the PDG-based and ranking-based $T$-scores, we refrain from drawing conclusions about the relative superiority of either method. Ultimately, the effectiveness of all fault localization algorithms will be assessed by end-users in practice.

We now compare \texttt{Sober} with \texttt{SliceChop}. In the study of \texttt{SliceChop} [38], the authors excluded the program \texttt{tcas} from the Siemens suite due to its small size, and the program \texttt{tot_info} because at that time their framework could not handle floating point operations. For the remaining five subject programs, which consist of 66 faulty versions in total, another 28 faulty versions were excluded for various reasons, leaving 38 versions used in the final evaluation. The authors reported
that for 23 out of the 38 versions, no more than 10.4% of the source code needed to be examined. We checked the quality of SOBER on the 66 versions of the five subject programs, and found that the T-score is less than 10.4% on 43 versions. Moreover, within the 38 faulty versions examined by SLICECHOP in [38], SOBER has a T-score of less than 8.4% on 27 versions. Because the ratio of examined code was not reported for each of the 38 versions in [38], no further comparison is performed here between SOBER and SLICECHOP.

Figure 3.9: Quality Degradation w.r.t. β%-Sampled Test Suite

3.4.5 Comparison from Other Perspectives

A comprehensive comparison between fault localization algorithms is hard, and many aspects must be considered for a fair comparison. For example, some important aspects are the runtime overhead, analysis complexity, localization accuracy and the accessibility of final fault localization reports. So far, we have been focusing on localization accuracy, and have demonstrated that SOBER is one of the most accurate algorithms. However, when compared on other aspects, SOBER might be inferior to other techniques, at least in its current state.

First, some techniques, like NN/PERM, Ct and SLICECHOP, only need one failing and multiple passing cases for fault localization, whereas SOBER, LIBLIT05 and TARANTULA, in principle need to collect the statistics from multiple failing cases. Secondly, SOBER could be inferior to LIBLIT05 in terms of the runtime overhead due to instrumentation. Specifically, since LIBLIT05 is based on the predicate coverage data, the instrumentation on a predicate can be disabled (in a similar way to
Jazz [73]) once the predicate has been evaluated. In contrast, SOBER needs to count the evaluation frequency throughout the execution. Finally, some algorithms, including TARANTULA, LIBLIT05 and delta debugging, have provided visual interfaces to increase their accessibility. Currently, no visual interface is available for SOBER, but one could be added in the future.

3.5 SOBER in an Imperfect World

Besides the probabilistic treatment of program predicates, there are two other factors that implicitly contribute to SOBER’s effectiveness as shown in Section 3.4. First, the test suite in our experiment is reasonably adequate given the program code size: Each subject program of the Siemens suite is accompanied by a few thousand test cases\(^1\). Intuitively, more reliable statistics can be collected from a more adequate test suite, and would enable SOBER to produce better fault localizations. Secondly, by taking the fault-free version as the test oracle, each execution is precisely labelled as either passing or failing. This provides SOBER with a noise-free analysis environment, which likely benefits SOBER’s inference abilities.

Although these two elements are highly desirable for quality localization, they are often not readily available in practice due to the potential high cost. For example, because the program specification varies from one component to another, exclusive test scripts for each component must be prepared by human testers. Although some tools can help expedite the generation of test cases [76] [17] [27], critical manual work is still unavoidable. Furthermore, besides the difficulty of test case generation, test oracle construction can be even more difficult. Again because of variations in program functionality, it is usually humans developers that prepare the expected outputs, or pass judgement about the correctness of outputs in practice.

Therefore, considering the difficulty of obtaining an adequate test suite and a test oracle, we regard the environment that we experimented with in Section 3.4 “a perfect world”. In order to shed some lights on how SOBER would work in practice, in this section we subject SOBER to an “imperfect world”, where adequate test suites and test oracles are not simultaneously available. Section 3.5.1 examines SOBER’s robustness to test inadequacy, and Section 3.5.2 studies how SOBER

\(^1\)In this article, we take the number of test cases as a rough measure of the test adequacy. More involved discussion about test adequacy is out of the scope of this study.
handles partially labelled test suites.

We regard, and hence believe, that the examination of SOBER in an “imperfect world” is both necessary and interesting. To some extent, this examination will bridge the gap between the perfect-world experiments (i.e., Section 3.4) and real-world practices that cannot be fully covered in any single research paper. We simulate the imperfect world with the 130 faulty versions of the Siemens suite. In parallel with SOBER, LIBLIT05 is also subjected to the same experiments for a comparative study, which illustrates how the two statistical debugging algorithms react to the imperfect world.

3.5.1 Robustness to Inadequate Test Suites

Because of the cost of an adequate test suite, people usually settle with inadequate but nevertheless satisfactory suites in practice. For instance, during the prototyping stage, one may not bother much with an all-around testing, and a preliminary test suite usually suffices. We now simulate an inadequate test suite by sampling (without replacement) the accompanying test suite of the Siemens suite. The sampled test suite becomes more and more inadequate as the sampling rate gets smaller.

Specifically, for each faulty version of the Siemens suite, we randomly sample a portion $\beta$ ($0 < \beta \leq 1$) of the original test suite $T$. Suppose $T$ consists of $N$ test cases, then \( \lceil N \ast \beta \rceil \) cases are randomly taken, constituting an $\beta$-sampled test suite, denoted as $T_\beta$. Because neither SOBER nor LIBLIT05 works when no failing cases are available, we repeat the above sampling until $T_\beta$ contains at least one failing case. Finally, both SOBER and LIBLIT05 are run on the same $T_\beta$ for each faulty program.

Fig. 3.9 plots how the quality varies with different sampling rates for both SOBER and LIBLIT05. We set $\beta$ equal to 100%, 10%, 1% and 0.1% respectively, so that $T_{100\%}$ represents the entire test suite and each of the following is roughly one tenth as small as the previous one. As $\beta$ gets smaller, the localization quality of both SOBER and LIBLIT05 gradually degrades. For example, in Fig. 3.9(a), curves for smaller $\beta$'s are strictly below those for higher sampling rates. A similar pattern for LIBLIT05 is also observed in Fig. 3.9(b). These observations are easily explainable. In statistical hypothesis testing, the confidence of either accepting or rejecting the null hypothesis is in general
proportional to the number of observations. Because SOBER bears a similar rationale to hypothesis testing, its quality naturally improves as more and more test cases are observed. Because Liblit05 relies on the accurate estimation of the two conditional probabilities, its quality also improves with more labelled test cases due to the Law of Large Numbers.

In Fig. 3.9(a), one can also notice that the curve for $\beta = 10\%$ is quite close to the highest. This suggests that SOBER obtains competitive results even when the test suite is only one tenth of the original. Moreover, Fig. 3.9 also indicates that even when $\beta$ is as low as $0.1\%$, both SOBER and Liblit05 are still consistently better than Ct. Based on the typical suite size from Table 3.1, $T_{0.1\%}$ contains at most six test cases, at least one of which is failing. As one can see, even with such an insufficient test suite, both SOBER and Liblit05 still outperform Ct. For example, without examining more than $20\%$ of the code, SOBER and Liblit05 locate $53.08\%$ and $51.54\%$ of the 130 faults respectively, while Ct works well with $38\%$ of the versions. This could be attributed to the underlying mechanism of Ct: It localizes faults by systematically contrasting the memory graphs of one passing and one failing execution. However, because the faults in the Siemens suite are mainly logic errors that rarely cause memory abnormalities, Ct has difficulties in identifying the “delta” and further locating the fault. On the other hand, because predicates express logic relations, it is no surprise that predicate-based algorithms work better.

Beside varying the sampling rate $\beta$, we also examined how the quality changed with respect to the absolute size of the test suite. However, because the size of the supplied test suite and the failing rate drastically vary from one faulty version to another, it makes little sense to set a uniform size for the test suite for quality examination. We therefore refrain from doing so, but choose instead to study how the number of failing cases could affect the localization quality, as described in the next subsection.

### 3.5.2 Handling Partially labelled Test Suites

Although an adequate test suite is difficult to obtain, preparing a test oracle that can automatically recognize each execution as either passing or failing is even more difficult. In some situations, test case generation can be relatively easy. For example, one can simply feed random strings to a program that consumes string inputs. However, these test cases are hardly useful until we know
what the expected outputs are.

In practice, except for programs that can be described by a program model, the expected outputs are usually prepared by human testers, either manually or assisted by tools. It is usually unrealistic for a tester to examine thousands of executions and label them. Instead, a tester will likely stop testing and return the faulty program to developers for patches when a small number of failing cases are encountered. At that time, the examined cases are labelled and the rest are unlabelled. This describes a typical scenario which exemplifies how partially labelled test suites arise in practice. In this subsection, we examine how well SOBER helps developers locate the underlying faults, when the test suite is partially labelled.

Formally, given a suite $T$, suppose a tester has examined and labelled a subset suite $T_e$ ($T_e \subseteq T$). Because manual labelling is usually expensive, it is common for $|T_e| \ll |T|$. Let $T_p$ and $T_f$ denote the set of passing and failing runs identified by the tester, then $T_e = T_p \cup T_f$ and $T_p \cap T_f = \emptyset$. We use $T_u$ to denote the unexamined part of the suite, i.e., $T_u = T - T_e$. $T$ is partially labelled if and
only if \( T_u \neq \phi \). The set relationship is further depicted in Fig. 3.10(a). The outer ellipse represents the entire test suite \( T \). The vertical line divides \( T \) into the full failing set \( T^f \) on the left and the full passing set \( T^p \) on the right. Certainly, \( T_f \subseteq T^f \) and \( T_p \subseteq T^p \). As one can see from Fig. 3.9(a), the best localization is achieved by SOBER when \( T_f = T^f \) and \( T_p = T^p \), i.e., when the given test suite \( T \) is fully labelled.

Now given the partially labelled test suite \( T \), the most straightforward scheme for SOBER is to analyze labelled test cases \( T_e \) only. Because \( T_e \) is fully labelled, SOBER can be immediately applied with \( T_e \). In fact, this scheme is equivalent to running SOBER on a \( \beta \)-sampled test suite, where \( \beta \approx \frac{|T_e|}{|T|} \). Because commonly \( |T_e| \ll |T| \), \( \beta \) is usually quite small. As a conservative estimation, \( \beta \) can be around 1%. In considering the Siemens programs, \( \beta = 1\% \) means that the tester examines tens, among thousands, of test cases, and identifies about five failing runs on the average. In our opinion, this can be a reasonable workload for the tester.

This scheme, although straightforward, does render reasonable localization results. As shown in Fig. 3.9(a), “SOBER 1% Sampled” is clearly better than CT. But it is also seen that a considerable gap exists between “SOBER 0.1% Sampled” and “SOBER 100% Sampled”. For concise reference, we use SOBER_Full to denote “SOBER 100% Sampled” in the following. Although the same quality as SOBER_Full is not (unrealistically) expected when \( T \) is partially labelled, we nevertheless believe that \( T_u \) can be utilized for better quality than with \( T_e \) only.

The above straightforward scheme apparently overlooks the information contained in \( T_u \). Although \( T_u \) does not provide any labelling information, their runtime statistics, if used properly, can assist SOBER in fault analysis. In this study, we restrict our discussion to reasonably developed programs that pass all but a few test cases. One can judge whether this assumption holds by examining the percentage of failing cases in \( T_e \). If, however, a program fails most cases in \( T_e \), the fault could be quite easy to find. For reasonably developed programs, we can choose to label all the unexamined test cases \( T_u \) as passing, and apply SOBER to the regarded failing and passing set \( T'_f \) and \( T'_p \), where \( T'_f = T_f \) and \( T'_p = T_p \cup T_u \). The difference between the two schemes is visualized in Fig. 3.10(b).

Let \( T_m \) represent the set of unexamined failing cases, i.e., \( T_m = T'_f - T_f \), then all the cases in \( T_m \) are mislabelled as passing in the above treatment. While this mislabelling unavoidably introduces
impurity into $T'_p$, the effect it has on SOBER is minimal: the $\theta_p$ calculated with $T'_p$ deviates negligibly from that with $T^u_p$ because $T'_p = T_p \cup T_u = T^u_p \cup T_m$ and $|T_m| \leq |T_f| \ll |T'_p|$.

On the other hand, by mislabelling $T_m$, we utilize the runtime statistics of the cases in $T'_p - T_p$, which are otherwise disregarded. In this way, $\theta_p$ can be estimated more accurately with $T'_p$ than with $T_p$ only. This could, in consequence, bring better localization quality. Therefore, this is essentially a tradeoff between grabbing more passing runs and (unavoidably) mislabelling some failing runs. In our belief, the gain from including more passing executions should surpass the loss from mislabeling. As will be shown shortly, this scheme achieves much better results than the straightforward scheme, and sometimes it even obtains comparable results to SOBER_Full.

We simulate partially labelled test suites using the Siemens programs. For each faulty version, we randomly select $m$ failing cases as $T_f$ (i.e., the set of failing cases identified by the tester). According to the above scheme, all the rest cases are regarded as passing, i.e., $T'_p$. We then run both SOBER and LIBLIT05 with the same $T'_p$ and $T'_f$ (recall that $T'_f = T_f$) for each of the 130 faulty versions. We experiment with $m$ equal to 1, 3, 5 and 10 respectively, and this represents the increasing effort that the tester puts into test evaluation. If a faulty version has less than $m$ failing cases, we take all the available failing cases. In the Siemens suite, there are 0, 4, 14, and 19 versions that have less than 1, 3, 5, and 10 failing cases. These versions were not excluded because they do represent real situations.

Fig. 3.11 plots the localization quality for both SOBER and LIBLIT05 with $m$ equal to 1, 3, 5 and 10 respectively. Curves for CT and SOBER_Full are also plotted as the baseline and ceiling quality in each subfigure. Among the four subfigures, Fig. 3.11(a) represents the toughest situation, where only one failing case is identified in each faulty version. This simulates a typical scenario where a developer starts debugging once a faulty execution is encountered. As expected, the quality of SOBER degrades considerably from SOBER_Full, but it is still better than CT.

For Fig. 3.9(a), we note that the $m = 1$ situation is at least as harsh as the situation with 0.1%-sampled test suites. Nevertheless, at least one failing run is in every 0.1%-sampled test suite. In order to demonstrate the effect of treating $T_u$ as passing, we re-plot the curve of SOBER with $\beta = 0.1\%$ in Fig. 3.11(a) with a dashed line. The remarkable gap between “SOBER” and “SOBER, 0.1%” suggests the benefit of treating unlabelled cases as passing.
The four subfigures of Fig. 3.11, viewed in a sequence, show that the quality of SOBER gradually improves as additional failing cases are explicitly labelled. Intuitively, the more failing cases that are identified, the more accurately the statistic $Y$ (Eq. 3.4) approaches to the true faulty behavior of predicate $P$, and hence the higher quality of the final predicate ranking list. LIBLIT05 also improves because of a similar reason.

3.5.3 Summary

In this section, we empirically examined how SOBER works in an imperfect world where either the test suite is inadequate or only a limited number of failing executions are explicitly identified. Our experiment demonstrates the robustness of SOBER under these harsh conditions. In addition, the scheme of tagging all unlabelled cases as passing is shown to effectively leverage SOBER’s quality.

3.6 Experimental Evaluation with Large Programs

Although the 130 faulty versions of the Siemens programs are appropriate for algorithm comparison, the effectiveness of SOBER nevertheless needs to be assessed with reasonably large programs. In this section, we report on the experimental evaluation of SOBER (with comparison with LIBLIT05) on nine faults in three median-sized programs. We report in Section 3.6.1 on the statistics of the four subject programs and the ten faults, and compare the debugging quality of SOBER and LIBLIT05 in Section 3.6.2. We then take a closed look at the grep, which demonstrates how the debugging result can help developers find the faults (Section 5.4.2). In order to prevent the illusion that SOBER is only applicable to semantic bugs, a detailed case study with bc is provided in Section 3.6.4, which demonstrates the effectiveness of SOBER in locating memory bugs. In particular, two faults are found from bc, and one of them has never been reported before.

3.6.1 Subject Programs and Seeded Faults

We obtained three subject programs, flex, grep, and gzip, together with their accompanying test suites from the “Software-artifact Infrastructure Repository” (SIR) [32], which “is a repository of software-related artifact meant to support rigorous controlled experimentation.” The exact version number, the number of physical Source Lines of Code (SLOC), and how many tests are included
Table 3.2: Characteristics of the Three Subject Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>SLOC</th>
<th>Number of Faults</th>
<th>Test Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex</td>
<td>2.4.7</td>
<td>9,212</td>
<td>5</td>
<td>525</td>
</tr>
<tr>
<td>grep</td>
<td>2.2</td>
<td>15,633</td>
<td>2</td>
<td>470</td>
</tr>
<tr>
<td>gzip</td>
<td>1.2.3</td>
<td>6,184</td>
<td>2</td>
<td>217</td>
</tr>
</tbody>
</table>

in the test suite are listed in Table 5.2. Especially, the program size (SLOC) is measured by the SLOCCount Tool\(^2\). The five faults in `flex` are injected by the SIR researchers, and the other four faults in `grep` and `gzip` are planted by ourselves because their accompanying test suites do not reveal any failing test cases.

![Figure 3.12: Five Faults in Flex-2.4.7](http://www.dwheeler.com/sloccount/)

We present the nine faults in `flex`, `grep` and `gzip` in Figures 3.12, 3.13 and 3.14, respectively, and readers can see that these faults are commonly made, and once made, are hard to find. Usually, developers need to trace a failing execution, and identify where the program behavior differs from what is expected. In the next subsection, we will present SOBER’s debugging result, and compare it with LIBLIT05.

\(^2\)http://www.dwheeler.com/sloccount/
3.6.2 An Overview of Debugging Quality and Comparison

In order to quantitatively (and hence objectively) compare the debugging results, we calculate the $T$-score for each of the nine faults based on the results from both SOBER and LIBLIT05. Table 3.3 summarizes the comparison of debugging quality between. The third column lists the fault type, which roughly describes the nature of the faults. The fourth column gives the number of failing cases over the total number of test cases in the accompanying test suite. Finally, the last two columns present the $T$-scores for both SOBER and LIBLIT05 in percentage. Figure 3.15 visualizes the comparison. The horizontal axis indexes the nine faults through simplified notation, e.g., “Flex Fault 1” is written as “Flex 1.” The vertical axis is for the $T$-score, and the lower the better because a lower score indicates less code needs to be examined. Figure 3.15 immediately reveals a few important observations about the debugging capacity of SOBER and LIBLIT05.

First, SOBER and LIBLIT05 are to some extent correlated although their methodologies are different as discussed in Section 3.3.7. For example, SOBER and LIBLIT05 achieve roughly the same debugging result for the 1st, the 4th, and the 5th faults in flex. This means that these three faults are equally difficult for SOBER and LIBLIT05. In particular, the 1st fault of flex is pretty
easy, and developers only need to examine 0.5% of the code with either SOBER and LIBLIT05. On the other hand, the 5th fault is much harder, and the T-score is 45.6% no matter either SOBER or LIBLIT05 is used. The 4th fault is mediocre, and roughly 15.4% of the code needs to be examined. In summary, there are certain faults that are equally difficult for both SOBER and LIBLIT05.

However, on the other side, besides the three faults on which both algorithms perform equally well, SOBER significantly outperform LIBLIT05 on all the rest six faults. For example, all the T-score is less than 10% for SOBER, while it is always over 20% for LIBLIT05. Especially, the contrast is striking for the four faults in grep and gzip, where SOBER always pinpoints the fault location or its close vicinity while LIBLIT05’s result is pretty far from the real fault location. Then it is natural to ask why LIBLIT05 performs so badly on these fault while it appears pretty good in the comparison with the Siemens suite (Section 3.4.3).

The explanation stems from the difference between SOBER and LIBLIT05 as discussed in Section 3.3.7, where we explained a scenario when LIBLIT05 will encounter difficulties because it only checks whether a predicate is ever evaluated true, and ignores the exact evaluation pattern inside each execution. We also pointed out there that if a predicate sits inside a loop, LIBLIT05 will likely
miss the predicate even if it is the most fault relevant predicate because the predicate can be ever evaluated as true in most passing and failing executions. A closed look at the four faults in grep (Figure 3.13) and gzip (Figure 3.14) confirms this reasoning: All the four faults are inside a particular loop, and those loops are frequently executed. This observation confirms our belief in the advantage of SOBER. But nevertheless we cannot claim the general superiority of SOBER over Liblit05 because no theoretical proof indicates this so far. There are a lot of other factors that determines the debugging quality, for example, the size, quality, and composition of test suites, besides the debugging algorithm.

### 3.6.3 A Detailed Case Study with Grep-2.2

In this subsection, we take a close look at how the debugging results from SOBER helps developers find the two faults in grep. Moreover, this case study also serve as an illustration of how to handle multiple faults with SOBER, although the $T$-scores presented in Table 3.3 are for each fault individually, i.e., each time one and only one fault is active.

The two faults are shown in Figure 3.13. The first fault (left) is an “off-by-one” error: an expression “+1” is appended to line 553 in the grep.c file. This fault causes failures on 48 of the 470 test cases. The second fault (right) is a “subclause-missing” error. The subclause ($lcp[i] == rcp[i]$) is commented out at line 2270 in file dfa.c. This fault incurs another 88 failing cases.

Although these two faults are manually injected, they do mimic realistic logic errors. Logic errors like “off-by-one” or “subclause-missing” may sneak in when developers are handling obscure corner conditions. Because logic errors, like these two, do not generally incur program crashes, they

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Failure Number</th>
<th>$T$-score (%)</th>
<th>SOBER</th>
<th>Liblit05</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-line 2.4.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault 1</td>
<td>Misuse $\geq$ for $&gt;$</td>
<td>163/525</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Fault 2</td>
<td>Misuse $=$ for $==$</td>
<td>356/525</td>
<td>1.6</td>
<td>22.3</td>
</tr>
<tr>
<td>Fault 3</td>
<td>Misuse $!=$ for $==$</td>
<td>69/525</td>
<td>7.6</td>
<td>22.3</td>
</tr>
<tr>
<td>Fault 4</td>
<td>Mis-parenthesize</td>
<td>22/525</td>
<td>13.4</td>
<td>15.4</td>
</tr>
<tr>
<td>Fault 5</td>
<td>Off-by-one</td>
<td>92/525</td>
<td>45.6</td>
<td>45.6</td>
</tr>
<tr>
<td>Grep 2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault 1</td>
<td>Off-by-one</td>
<td>48/470</td>
<td>0.3</td>
<td>40.8</td>
</tr>
<tr>
<td>Fault 2</td>
<td>Subclause-missing</td>
<td>88/470</td>
<td>0.2</td>
<td>51.7</td>
</tr>
<tr>
<td>Gzip 1.2.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault 1</td>
<td>Subclause-missing</td>
<td>65/217</td>
<td>0.05</td>
<td>21.3</td>
</tr>
<tr>
<td>Fault 2</td>
<td>Subclause-missing</td>
<td>17/217</td>
<td>3.0</td>
<td>41.7</td>
</tr>
</tbody>
</table>

Table 3.3: Debugging Quality Evaluated on Large Programs
are usually harder to debug than those causing program crashes. In the following, we illustrate how SOBER helps developers find these two faults.

We first instrument the source code. According to the instrumentation schema described in Section 5.4.1, grep is instrumented with 1732 branch and 1404 return predicates. The first run of SOBER with the 136 failing (due to the two faults) and the remaining 334 passing cases produces a predicate ranking, with the top-3 predicates are listed in Table 3.4. For easy reference, the three predicates are also marked at their instrumented locations in Figure 3.13.

As we can see, the predicates $P_{1470}$ and $P_{1484}$ point to the faulty function for the first fault. The predicate $P_{1470}$ is 4 lines above the real fault location. The predicate $P_{1952}$, on the other hand, points directly to the exact location of the second fault. Now let us pretend the faults are unknown, and explore how these top predicates help developers locate the faults.

Given the top-ranked predicates, it is natural to ask why they are ranked high. We find that
Table 3.4: Top Three Predicates from the First Run of SOBER

<table>
<thead>
<tr>
<th>Ranks</th>
<th>Filename</th>
<th>Line Num.</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{1470}$</td>
<td>grep.c</td>
<td>549</td>
<td>(lastout) == true</td>
</tr>
<tr>
<td>$P_{1484}$</td>
<td>grep.c</td>
<td>574</td>
<td>(beg != lastout) == true</td>
</tr>
<tr>
<td>$P_{1952}$</td>
<td>dfa.c</td>
<td>2270</td>
<td>(lcp[i] != '0') == true</td>
</tr>
</tbody>
</table>

the sample mean and standard deviation of the evaluation bias of $P_{1470}$ (denoted by $\pi(P_{1470})$) are 0.90 and 0.25 in the 136 failing cases, but are 0.99 and 0.065 in the rest 344 passing cases. This immediately suggests that $P_{1470}$ is mostly evaluated true in passing cases with a small variance, but is mostly evaluated false in some failing cases, as indicated by its much larger variance in failing cases. By examining $\pi(P_{1470})$ in the failing cases, we find that $\pi(P_{1470})$ is smaller than 0.1 in five failing cases. Therefore, we know that $P_{1470}$ can be evaluated mostly as false in these failing cases, whereas it is mostly true in passing cases. Similarly, we find that $P_{1484}$ is considerably evaluated as true in failing cases, but mostly false in passing cases.

We notice that when $P_{1484}$ evaluates true, the variable lastout is reset to 0, which immediately causes $P_{1470}$ to evaluate as false in the next iteration. This explains why predicates $P_{1470}$ and $P_{1484}$ are both ranked at the top. In order to find why “beg != lastout” tends to evaluate to true in failing cases, a developer would pay attention to the assignment to variables beg and lastout. Within the for loop from lines 541 to 580, there are no other assignments to lastout besides lines 549 and 575. Then, the developer would examine lines 553, 566 and 571, where beg gets assigned. A developer familiar with the code will then identify the fault.

After fixing the first fault, a second run of SOBER with the 88 failing and 382 passing cases puts $P_{1952}$ at the top. A developer paying more attention to line 2270 of the dfa.c file would find the fault, as $P_{1952}$ points to the exact fault location. Because SOBER is only for fault localization, it is the developer’s responsibility to confirm the fault location, and fix it. To the best of our knowledge, no tools can automatically suggest patches for logic errors without assuming any specifications.

3.6.4 Effectiveness with Memory Faults: A Case Study with Bc 1.06

While all the discussion so far focuses on the debugging quality with semantic bugs or logic errors, readers should not have the illusion that SOBER is only inapplicable with memory bugs, i.e., bugs that cause program crashes rather than “silent deaths”. In principle, SOBER relies on no particu-
larity that is exclusive to semantic bugs. In this subsection, we report on a case study of SOBER with a real-world program bc, on which SOBER identified two buffer overflow faults, one of which had never been reported before.

The program bc is a calculator program that accepts scripts written in the bc language, which supports arbitrary precision calculations. We used the 1.06 version of the bc program that is shipped with most recent UNIX/Linux distributions. It has 14,288 LOC and a buffer overflow fault has been reported in [58][59].

This experiment was conducted on a P4 3.06 GHz machine running Linux RedHat 9 with gcc 3.3.3. Inputs to bc 1.06 are 4,000 valid bc programs that were randomly generated with various size and complexity. We generated each input program in two steps: First, a random syntax tree was generated in compliance with the bc language specification; second, a program was derived from the syntax tree.

With the aid of SOBER, we quickly identified two faults in bc 1.06, including one that had never been reported. Among the 4,000 input cases we tested on bc 1.06, there were 3,479 correct cases and 521 incorrect ones. After running through these test cases, the analysis from SOBER reported “indx < old_count” as the most fault-relevant predicate. This predicate pointed to the variable old_count in Line 137 of storage.c (shown in Fig. 3.16). A quick scan of the code showed that old_count copied its value from v_count. By putting a watch on v_count, we found that v_count was overwritten when a buffer named genstr overflowed (in bc.y, Line 306). The buffer genstr was 80-byte-long, and was used to hold byte code characters. An input containing complex and

```c
void more_variables ()
{
    ...
    127   old_count = v_count;
    ...
    137   for (indx = 3; indx < old_count; indx++)
    ...
    141   for (; indx < v_count; indx++)
    ...
}
```

Figure 3.16: First Fault in bc 1.06, in storage.c
relatively large functions could easily overflow it. To the best of our knowledge, this fault had never been reported before. We manually examined the statistics of the top-ranked predicate, and found that its evaluation bias in correct and incorrect executions was 0.0274 and 0.9423 respectively, which intuitively explained why SOBER worked. LIBLIT05 also ranked this predicate at the top.

After we fixed the above fault, a second run of SOBER (3303 correct and 697 incorrect cases) generated a fault report with the top predicate “a_count < v_count”, which pointed to Line 176 of storage.c (shown in Fig. 3.17). This was most likely a copy-paste error where a_count would have been used in the position of v_count. This fault had been reported in previous studies [58] [59].

As a final note, predicates identified by SOBER for these two faults are far from the actual crashing points. This suggests that SOBER picks up predicates that characterize the scenario under which faults are triggered, rather than merely the crash site.

3.6.5 Summary

In this section, we reported on the debugging quality of SOBER with four median-sized programs containing 11 faults, which clearly demonstrates the effectiveness of SOBER in locating memory bugs, as well as semantic bugs. In particular, the comparison in Section 3.6.2 reveals the advantages of SOBER over LIBLIT05 for faults with a common pattern: faults embedded in frequently executed loops. Although we did not encounter any faults (among the examined 11 faults) on which LIBLIT05 significantly outperforms SOBER, we cannot exclude such possibility. We leave the choice of debugging algorithms to the ultimate users.
3.7 Discussion

3.7.1 Related Work

In this section, we briefly review previous work related to fault detection in general. Static analysis techniques have been used to verify program correctness against a well-specified program model [9, 22], and to check real codes directly for Java [90] and C/C++ programs [75]. Engler et al. [33] further show that the correctness rules sometimes can be automatically inferred from source code, hence saving, to some extent, the cost of preparing specifications. Complementary to static analysis, dynamic analysis focuses more on the runtime behavior, and often assumes fewer specifications. SOBER belongs to the category of dynamic analysis.

Within dynamic analysis, most fault localization techniques are based on the contrast between failing and passing cases [6, 79, 95, 23, 81, 47, 58, 59, 64]. For example, invariants that are formed from passing cases can suggest potential fault locations if they are violated in any failing cases [79]. Readers interested in the details of invariants are referred to the project DAIKON [34]. The DIDUCE project [39] monitors a more restricted set of predicates and relaxes them in a similar manner to DAIKON at runtime. After the set of predicates becomes stable, the DIDUCE tool relates further violations as indications of potential faults. This approach is demonstrated to be effective on four large software systems. However, as invariants are a special kind of predicates that hold in all passing runs, they may not be effective in locating subtle faults as suggested by Pytlik and Renieris in [79]. In comparison, the probabilistic treatment of predicates implemented by SOBER naturally relaxes this requirement and is shown to achieve much better localization results on the Siemens suite.

Contrasts based on program slicing [87] and dicing [66] are also shown effective for fault localization. For example, Agrawal et al. [6] present a fault localization technique, implemented as \( \chi \) slice, which is based on the execution traces of test cases. This technique displays and contrasts the dices of one failing case to those of multiple passing cases. Jones et al. [46] describe a similar approach implemented as TARANTULA. Unlike \( \chi \) slice, TARANTULA collects the testing information from all passing and failing cases, and colors suspicious statements based on the contrast. Later, Renieris and Reiss [81] find that the contrast renders better fault localization when the given failing case is
contrasted with the most similar passing case (i.e., the nearest neighbor). In comparison, SOBER collects the evaluation frequency of instrumented predicates, a much richer information base, and quantifies the model difference through a statistical approach.

While all the fault localization algorithms examined in this article are designed for programming professionals, recent years have also witnessed an emergence of fault localization algorithms especially tuned to assist end users in fault diagnosis. For example, Ayalew and Mittermeir propose a technique to trace faults in spreadsheets based on “interval testing” and slicing [10]. Ruthruff et al. improve this approach by allowing end-users to interactively adjust their feedbacks [83]. The Whyline prototype realizes a new debugging paradigm called “interrogative debugging”, which allows users to ask why did and why didn’t questions about runtime failures [50].

The power of statistical analysis is demonstrated in program analysis and fault detection. Dickinson et al. find program failures through clustering program execution profiles [30]. Their subsequent work [31] first performs feature selection using logistic regression and then clusters failure reports within the space of selected features. The clustering results are shown to be useful in prioritizing software faults. Early work of Liblit et al. on statistical debugging [58] also adopts logistic regression in sifting predicates that are correlated with program crashes. In addition, they impose $L_1$ norm regularization during the regression so that predicates that are really correlated are distinguished. In comparison, our method SOBER is a statistical model-based approach, while the above statistical methods follow the principle of discriminant analysis. Specifically, SOBER features a hypothesis testing-like approach, which has not been seen previously in the fault localization literature.

### 3.7.2 Threats to Validity

Like any empirical study, threats to validity should be considered in interpreting the experimental results presented in this article. Specifically, the results obtained with the Siemens suite cannot be generalized to arbitrary programs. However, we expect that on larger programs with greater separation of concerns, most fault localization techniques will do better. This expectation is supported by existing studies with CT, LIBLIT05 and TARANTULA [23, 59, 47], as well as the experiments in Section 3.6 of this study.
Threats to construct validity concern the appropriateness of the quality metric for fault localization results. In this article, we adopt the PDG-based $T$-score, which was proposed Renieris and Reiss. Although this evaluation framework involves no subjective judgements, it is by no means a comprehensively fair metric. For instance, this measure does not take into account how easily a developer can make sense of the fault localization report. Recent work [23] also identifies some other limitations of this measurement. In previous work, a ranking-based $T$-score is used to evaluate the effectiveness of Tarantula. Although both forms of $T$-score estimate the human efforts needed to locate the fault, it is yet unclear whether they agree. The comparison of Tarantula with other algorithms in Section 3.4.4 assumes the equivalence between the two forms. More extensive studies are needed to clarify this issue.

Finally, threats to internal validity concern the experiments of Sober with the programs grep and bc, discussed in Section 3.6. Specifically, the two logic errors in grep are injected by ourselves. However, because these two logic errors do not incur segmentation faults, they are generally harder to debug than faults that do, even for human beings. In contrast, most previous cases studies target crashing faults [95, 23, 58, 59]. Therefore, the experiment with grep demonstrates the effectiveness of Sober on large programs with logic errors. In order to minimize the threats to external validity about experiments with large programs, a case study with bc is also presented, which illustrates the effectiveness of Sober with real faults. However, two experiments are still insufficient to make claims about the general effectiveness of Sober on large programs. Ultimately, all fault localization algorithms should be subjected to real practice, and evaluated by end-users.

3.8 Conclusions

In this article, we proposed a statistical approach to localize software faults without prior knowledge of program semantics. This approach tackles the limitations of previous methods in modeling the divergence of predicate evaluations between correct and incorrect executions. Systematic evaluations through the Siemens suite, together with two case studies with grep and bc, clearly demonstrated the advantages of our method in fault localization. We also simulated an “imperfect world” to investigate Sober’s robustness to the harsh scenarios that may be encountered in practice. Our experiment result favorably supports Sober’s applicability.
Chapter 4
Statistical Debugging-Aided Failure Triage

4.1 Introduction

Failure proximity is one of the central questions in automated failure prioritization and diagnosis. Nowadays, as most complex software systems, like Netscape/Mozilla, Microsoft Windows and GNOME, feature an automated failure report component, a huge number of failing traces (i.e., failures) are collected every day. In order to prioritize fault diagnosis, failing traces due to the same fault are expected to be clustered together. Lying in the center of the clustering problem is the definition of failure proximity, i.e., what traces should be regarded as similar so that failures due the same fault are clustered together.

The optimal proximity is based on the fault each failure is due to: Failures due to the same fault are defined similar to each other. However, as the real fault for each failure is unknown without expensive manual investigation, the optimal proximity is usually unreachable. As an alternative, previous work [78] defines failure proximity based on the literal trace similarity, which we call T-PROXIMITY. T-PROXIMITY regards two failures as similar if they exhibit similar behaviors, for example, similar branch or statement coverage. T-PROXIMITY approximates the optimal proximity by hypothesizing that the same fault incurs similar behaviors.

Although T-PROXIMITY is widely adopted to measure the proximity between executions [30, 31, 78], it is nevertheless weak in characterizing the semantic proximity between failing traces. First, because a fault can be triggered with different inputs, failing traces due to the same fault can be, and usually are, divergent. This suggests that clustering based on T-PROXIMITY tends to group together failures exhibiting similar behaviors, rather than those due to the same fault. Second, clustering based on T-PROXIMITY does not provide any information about the possible fault location for each failure cluster. In consequence, a developer needs to manually investigate
each cluster, and assign failing traces in each cluster to appropriate developers.

In order to circumvent the above weaknesses of T-Proximity, a new definition of failure proximity is proposed in this paper: Instead of relying on trace similarity, we regard two failures as similar if they suggest roughly the same fault location. The motivation is that now that it is optimal to cluster together failures due to the same fault, why do not we first find the likely fault location for each failure automatically, and then cluster failures according to the found fault locations? We demonstrate that this is a viable approach, and the statistical debugging tool SOBER[64] can be used to find the likely fault location for each failure. Because the fault location found through SOBER is in the form of a predicate ranking, we name thus defined proximity rank proximity, or R-Proximity in short. The basic idea of R-Proximity is explained below.

Suppose \( m \) failing traces, \( F = \{f_1, f_2, \ldots, f_m\} \), and \( n \) passing traces, \( P = \{p_1, p_2, \ldots, p_n\} \), are collected, and each trace is represented by the evaluation history of instrumented predicates. Conventionally, SOBER takes \( F \) and \( P \) as inputs, and generates a ranking \( \tau \) of all instrumented predicates, i.e., \( \tau = \text{SOBER}(F, P) \). The predicate ranking \( \tau \) is the debugging result, and its top predicates likely point to the fault location or the fault vicinity. An important observation is that neither \( F \) nor \( P \) is required in their entirety in the application of SOBER; instead, any subsets of \( F \) and \( P \) can be fed into SOBER for fault localization. As an extreme case, we can contrast each failing trace \( f_i \) against \( P \), and find the fault location \( \tau_i \) each \( f_i \) suggests automatically, i.e., \( \tau_i = \text{SOBER}(\{f_i\}, P), \forall f_i \in F \). We call \( \tau_i \) the fault-aware fingerprint of \( f_i \) because it embodies the fault location \( f_i \) suggests. Based on the fault-aware fingerprints, failure proximity can be measured by examining how \( \tau_i \)’s agree with each other: Two failing cases are regarded as similar, if their suggested fault locations roughly agree. A weighted form of the Kendall’s tau distance is developed in this article (Section 4.3.3) to quantify the agreement. The connection and difference between R-Proximity and T-Proximity are illustrated in Figure 4.1.

In comparison with T-Proximity, R-Proximity is semantically closer to the optimal proximity because it is based on the likely fault location each failure suggests. The SOBER algorithm is used here to substitute for the otherwise expensive manual investigation. As a result, clustering under the R-Proximity tends to be more reasonable.

Moreover, as R-Proximity is defined on the fault location each failing trace suggests, clus-
Figure 4.1: T-Proximity and R-Proximity

terings under R-PROXIMITY automatically provide a guess about the fault location for each cluster. Specifically, because only failures agreeing on the fault location are clustered together, their agreement represents the fault location for this cluster. In consequence, failures in this cluster can be assigned to appropriate developers automatically. In contrast, since clusters formed under T-PROXIMITY do not bare such fault information, manual investigation is usually needed for failure assignment.

Finally, R-PROXIMITY can also help developers interpret statistical debugging results, and this, again, cannot be accomplished with T-PROXIMITY. Previous work has established the accuracy of statistical debugging [58, 59, 64, 62], but despite its general accuracy, developers usually complain about the interpretability of statistical debugging. In the first place, developers would wonder about the trustworthiness of the debugging result τ, i.e., to what extent the debugging result can be trusted. As a matter of fact, the lack of trustworthiness estimation is a common problem for most, if not all, fault localization techniques. In the second place, even when developers fully trust the debugging result, developers may find τ hard to understand without a concrete failing case. For instance, they may wonder why certain predicates are ranked high, and how they relate to the underlying faults. Under R-PROXIMITY, because the proximity between τ and each failure can be calculated, the quality of τ can be assessed before its content is examined. If τ is assessed as accurate, a concrete failing trace that best interprets τ can be selected to help developers utilize the debugging result. On the other hand, if τ is assessed as less accurate, developers can simply ignore the debugging result, and resort to other approaches. In section 4.5, we report on two case studies to illustrate the two possibilities, as well as the other two advantages.

Admittedly, R-PROXIMITY is nonetheless an approximation to the optimal proximity, and the
approximation relies on the debugging accuracy of SOBER. In this study, SOBER performs reasonably well. In the future, when better statistical debugging algorithms are developed, SOBER can be safely replaced without changing the definition of R-PROXIMITY. More generally, the idea of defining failure proximity based on fault localization results is compatible with other automated debugging algorithms, like delta debugging [95] and TARANTULA[47]. What one needs for a particular debugging algorithm is a matching distance metric to assess the proximity between debugging results.

In summary, this article makes the following contributions:

1. A new definition of failure proximity, namely R-PROXIMITY, is proposed, which leverages existing statistical debugging algorithms. Because R-PROXIMITY is defined on the fault location each failing trace suggests, it is semantically closer to the optimal proximity than existing T-PROXIMITY is.

2. The core technique of R-PROXIMITY is the fault-aware fingerprinting, which transforms each failing trace into a predicate ranking through statistical debugging. To the best of our knowledge, this is the first piece of work that utilizes statistical debugging for failure investigation. This work suggests that statistical debugging, or more generally, fault localization techniques are not restricted to fault localization.

3. Besides the fault-aware fingerprinting, a weighted form of the Kendall’s tau distance is proposed in the definition of R-PROXIMITY. It is proposed to accommodate the specialities of predicate rankings, and it proves appropriate and effective in this study.

4. Two detailed case studies with grep-2.2 and gzip-1.2.3 are reported to exemplify the unique advantages of R-PROXIMITY over T-PROXIMITY. In addition, a set of experiments are also carried out to validate the design of the weighted Kendall’s tau distance.

The rest of this article is organized as follows. Section 4.2 provides some preliminary knowledge and introduces terminologies for the development of this article. The technical details of R-PROXIMITY are explained in Section 4.3. We discuss the advantages of R-PROXIMITY over T-PROXIMITY in Section 4.4, and report on two case studies to support the discussion in Section 63.
4.5. The set of experiments validating the design of the weighted Kendall’s tau distance are presented in Section 4.6. With related work and threats to validity discussed in Section 5.5, Section 5.6 concludes this article.

4.2 Preliminaries and Terminologies

Failure proximity is not a new topic in software engineering research, although it is first given the name in this article. In fact, the problem of how to define and measure the similarity between executions is one of the central questions in many researches related to dynamic analysis [16, 30, 78, 40]. Before a proximity is defined, executions are first profiled according to the runtime behaviors, such as control flows and statement coverage. Because most runtime behaviors can be expressed in terms of predicates, we assume that an execution is profiled as a predicate vector in this article, and the predicate vector is referred to as the execution trace, or trace in short.

Suppose a program \( P \) is instrumented with \( L \) predicates, then the execution with input \( t \) is profiled as an \( L \)-dimensional vector \( v_t \), where the \( i \)th dimension \( v_t(i) \) records how many times the \( i \)th predicate \( P_i \) evaluates true during the execution. Depending on the need and the overhead tolerance, many kinds of predicates can be instrumented. In this article, we instrument subject programs with two kinds of predicates as shown below. They have been shown effective in characterizing program behaviors in previous studies [59, 64].

- [boolean]: For each boolean expression \( B \), a predicate “\( B=\text{true} \)” is instrumented.

- [return]: For each function call site, three predicates, “\( R>0 \)” , “\( R=0 \)” and “\( R<0 \)” are instrumented, where \( R \) is the function return value.

With execution traces profiled as predicate vectors, the pari-wise distances between failing traces can be calculated with any existing or newly defined distance metrics, such as the Euclidean distance. A set of pair-wise distances determines a proximity between failing traces. When the set of distances is calculated from traces directly, the resultant proximity is T-PROXIMITY. In particular, we use T-PROXIMITY to refer to the proximity defined with the Euclidean distance in this article. In Section 4.3, we examine R-PROXIMITY, which, in contrast, calculates the distances indirectly from traces.
Before delving into the technical details of R-PROXIMITY, a few terminologies are introduced here. We say a test case \( t \) fails on the program \( \mathcal{P} \), or equivalently, the faults in the program \( \mathcal{P} \) fail the test case \( t \), if and only if the output is different from what is expected. If it is, \( t \) is a failing case, and its execution trace is a failing trace or a failure. On the other hand, if the output of \( t \) is the same as what is expected, \( t \) passes the program \( \mathcal{P} \), \( t \) is a passing case, and its execution trace is a passing trace.

A statistical debugging algorithm, like SOBER, generates a ranking \( \tau \) of all the instrumented \( L \) predicate as the fault localization result (or the debugging result). Let \( \tau(P_i) \) denote the rank of predicate \( P_i \) in \( \tau \), then we say predicate \( P_i \) ranks higher or before predicate \( P_j \) if and only if \( \tau(P_i) < \tau(P_j) \), and the predicate \( P_i \) is ranked within top-\( k \) if and only if \( \tau(P_i) \leq k \). For example, if a predicate is the second highest in \( \tau \), its rank is 2, and it is ranked within top-\( k \) (\( 2 \leq k \leq L \)), but out of top-1.

### 4.3 R-Proximity: Failure Proximity Defined via Statistical Debugging

We now discuss the technical details of R-PROXIMITY. There are two core techniques in the definition of R-PROXIMITY: fault-aware fingerprinting and weighted Kendall’s tau distance. They are discussed in Sections 4.3.1 and 4.3.3, respectively. The discussion on the traditional Kendall’s tau distance in Section 4.3.2 is to ease the development of the weighted form.

#### 4.3.1 Fault-aware Fingerprinting

Given \( m \) failing traces \( F = \{f_1, f_2, \ldots, f_m\} \) that are due to \( r \) faults \( B = \{b_1, b_2, \ldots, b_r\} \), there exist a “due-to” relationship between the \( m \) failures and the \( r \) faults. Under the optimal proximity, \( f_i \)'s due to the same fault have small distances in between, and failures due to different faults have larger ones.

Given that the optimal proximity is unachievable, we wonder whether it is possible to approximate the optimal through automated debugging. Specifically, we can find the likely fault location for each failure automatically. If the likely fault location well approximates the real fault location,
a sub-optimal proximity can be defined accordingly. In this way, the expensive manual investigation as required by the optimal proximity is substituted by automated debugging, which totally unloads human beings. Given the accuracy of statistical debugging as witnessed in previous studies [58, 59, 64, 47, 62], we believe this is a viable approach.

In this study, we use SOBER [64], an existing statistical debugging tool, to find the likely fault location for each failure. SOBER localizes underlying faults by contrasting the set of failing trace $F$ against a set of passing traces $P = \{p_1, p_2, \ldots, p_n\}$. It contrasts the evaluation bias of each predicate in failing traces against that in passing ones. The evaluation bias basically measures, for each predicate and in each execution, what percentage of evaluations are true. For example, if one predicate $P$ is evaluated 10 times during one execution, and it evaluates true for three times, the evaluation bias of $P$ in this execution is 0.3. Readers interested in the details about SOBER are referred to [64].

Conventionally, SOBER takes $P$ and $F$ as inputs, and produces a predicate ranking $\tau$ as the debugging result, i.e., $\tau = \text{SOBER}(F, P)$. As $\tau$ is derived from all failing traces, it is called the global ranking. Usually, higher ranked predicates in $\tau$ are more likely to be fault-relevant, i.e., pointing to the fault location or the vicinity.

As one may have noticed, SOBER is not restricted to contrasting $F$ against $P$ in their entirety. Instead, any subsets of $F$ and $P$ can be contrasted for fault localization. As an extreme scenario, each failing trace $f_i \in F$ can be contrasted against $P$, generating a corresponding debugging result $\tau_i$, i.e.,

$$\tau_i = \text{SOBER}({f_i}, P) \ (i = 1, 2, \ldots, m). \quad (4.1)$$

The $\tau_i$'s are called individual rankings because they are derived from each failing traces individually.

Although $\tau_i$'s may not pinpoint the exact fault location for each failing trace (depending on the quality of SOBER, $f_i$ and $P$), they do suggest what predicates $f_i$'s regard as fault-relevant respectively. In this sense, $\tau_i$ embodies $f_i$'s opinion on possible fault locations, and this is why the transformation described by Eq. 4.1 is called the fault-aware fingerprinting. By virtue of the fault-aware fingerprinting, the failure proximity can be measured in a predicate ranking space. Precisely, two failing traces are defined close to each other if the individual rankings derived from them suggest roughly the same fault location. To this end, a proper distance definition is needed.
to quantify to what extent two predicate rankings suggest the same fault location. We discuss this problem in the following two subsections. Because this proximity is defined in the ranking space, it is called “rank-proximity”, or R-PROXIMITY in short.

4.3.2 Classic Kendall’s tau Distance

Ultimately, we will use a weighted Kendall’s tau distance to quantify the agreement between two predicate rankings. In order to see why a weighted form is needed, we first introduce the classic form Kendall’s tau distance in this subsection.

The classic Kendall’s tau distance is widely adopted to quantify the disagreement between two rankings [49]. Let $\pi$ and $\sigma$ be two rankings of the $L$ predicates, the Kendall’s tau distance $D_K(\pi, \sigma)$ is defined as

$$D_K(\pi, \sigma) = \sum_{1 \leq i < j \leq L} K(P_i, P_j)$$

(4.2)

where

$$K(P_i, P_j) = \begin{cases} 1 & \text{if } [\pi(P_i) - \pi(P_j)][\sigma(P_i) - \sigma(P_j)] < 0, \\ 0 & \text{otherwise}. \end{cases}$$

Predicates $P_i$ and $P_j$ constitute a discordant pair if their relative orders in $\pi$ and $\sigma$ disagree, and the Kendall’s tau distance essentially counts the number of discordant pairs between $\pi$ and $\sigma$.

Although the Kendall’s tau distance is a valid and reasonable distance measure for rankings in general, it is nevertheless short in quantifying the distance between predicate rankings. In the first place, most instrumented predicates are superfluous, and these superfluous predicates should be excluded in distance computation. Specifically, we need to find the subset of fault-relevant predicates, and project original $\tau_i$’s onto the subset. Let $S$ be the set of instrumented predicates ($L = |S|$), and $S_r \subseteq S$ is the subset of fault-relevant predicates ($l = |S_r|$), a ranking $\tau$ is projected onto $S_r$ if predicates out of $S_r$ are removed from $\tau$. We use $\tau'$ to denote the projected ranking of $\tau$. In the second place, even within $S_r$, all predicates are not equally fault-relevant. In general, we expect discordant pairs of more relevant predicates contribute more to the final ranking distance. We use the following example to illustrate the two points.

Example 1 Suppose the program $\mathcal{P}$ is instrumented with 6 predicates, i.e., $S =$
\{P_1, P_2, P_3, P_4, P_5, P_6\}, and three failing traces are collected. Figure 4.2 shows the global ranking \(\tau\) and individual ranking \(\tau_1, \tau_2, \tau_3\). For legibility, predicate indices are used in rankings. Clearly, predicates are not equally fault-relevant. For example, \(P_2\) is more fault-relevant than \(P_5\) because it ranks higher in all rankings. Furthermore, it is easy to calculate that \(D_K(\tau_1, \tau_2) = D_K(\tau_1, \tau_3) = 2\), but intuitively, we would expect that \(\tau_1\) is closer to \(\tau_3\) than to \(\tau_2\), because \(\tau_1\) and \(\tau_3\) only disagree on the relative orders between less relevant predicates.

Therefore, in order to reasonably quantify the distance between predicate rankings, predicates need to be properly weighted based on their fault relevance. Less relevant predicates gain a small or zero weights, and are then discounted or ignored in the final distance calculation. We discuss predicate weighting and the weighted Kendall’s tau distance in the next subsection.

4.3.3 Weighted Kendall’s tau Distance

Because no guidance is available for setting predicate weights in general, we choose to automatically derive the predicate weights from the global and individual rankings. Intuitively, top predicates in these rankings are likely fault-relevant.

First, the top-\(k_1\) predicates of \(\tau\) are taken as fault-relevant, and the weight of predicate \(P_i\) is determined by

\[
W_{k_1}^1(P_i) = \frac{I(k_1 - \tau(P_i))}{k_1},
\]

where \(I(x)\) is an indicator function that equals to 1 if \(x \geq 0\) and 0 otherwise. If a predicate ranks lower than \(k_1\) in \(\tau\), it gets a zero weight, and is not selected into \(S_\tau\) at this step. For those selected predicates in \(S_\tau\), equal weights are assigned although one could also assign decaying weights to lower-ranked predicates.
Second, the individual rankings $\tau_i$’s also suggest fault-relevant predicates. Intuitively, the $m$ individual rankings are like $m$ votes for fault-relevant predicates, and in consequence, predicates favored by more rankings are likely fault-relevant. Therefore, if the top-$k_2$ predicates of each ranking are considered, the weight of predicate $P_i$ is defined as

$$W_2^{k_2}(P_i) = \frac{\sum_{j=1}^{m} I(k_2 - \tau_j(P_i))}{mk_2}, \quad (4.4)$$

where $I(x)$ is the same indicator function as that in Eq. 4.3. This is called the frequency weighting because the weight is proportional to in how many $\tau_i$’s $P_i$ ranks within the top-$k_2$.

Combining these two components, the weight of predicate $P_i$ is

$$W(P_i) = (1 - \alpha)W_1^{k_1}(P_i) + \alpha W_2^{k_2}(P_i), \quad (4.5)$$

where $\alpha$ is the parameter balancing the two components. In fact, predicates within neither the top-$k_1$ of $\tau$ nor the top-$k_2$ of $\tau_i$’s receive a zero weight, and are excluded in distance calculation. Predicates with a nonzero weight then constitute the set of fault-relevant predicates $S_r$. By default, $k_1$, $k_2$ and $\alpha$ are set as 10, 1 and 0.1, respectively. We will discuss the roles played by each parameter in Section 4.6 through experiments.

**Example 2** Continue Example 1. With $k_1 = 4$, $k_2 = 1$ and $\alpha = 0.1$, $W_1 = (0.25, 0.25, 0, 0.25, 0.25, 0)$, $W_2 = (0.67, 0.33, 0, 0, 0, 0)$, and $W = (0.292, 0.258, 0, 0.25, 0.25, 0)$. Therefore, predicates $P_3$ and $P_6$ are excluded, and $S_r = \{P_1, P_2, P_4, P_5\}$.

When predicate weights have been derived, the weighted Kendall’s tau distance is defined as below.

**Definition 3 (Weighted Kendall’s tau Distance)** Given $\pi$ and $\sigma$ two rankings of the $L$ predicates, the weighted Kendall’s tau distance $D_{W,K}$ is

$$D_{W,K}(\pi, \sigma) = \sum_{1 \leq i < j \leq L} K(P_i, P_j)W(P_i, P_j), \quad (4.6)$$

where $K(P_i, P_j)$ is the same as that in Eq. 4.2, and $W(P_i, P_j) = W(P_i)W(P_j)$. 69
As the weighted Kendall’s distance is used to measure the distance between predicate rankings, its validity as a distance metric is proved by the following theorem.

**Theorem 2 (Metric Validity)** The weighted Kendall’s tau distance \( D_{W,K} \) is a metric on the set of \( L \)-predicate rankings if \( W(P_i, P_j) > 0 \) for \( 1 \leq i < j \leq L \), i.e., let \( \pi, \sigma, \eta \) be three rankings of \( L \) predicates, the following four properties hold:

1. \( D_{W,K}(\pi, \sigma) \geq 0 \),
2. \( D_{W,K}(\pi, \sigma) = 0 \) iff \( \pi = \sigma \),
3. \( D_{W,K}(\pi, \sigma) = D_{W,K}(\sigma, \pi) \),
4. \( D_{W,K}(\pi, \sigma) \leq D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma) \).

The proofs for (1), (2) and (3) are trivial once \( K(P_i, P_j) \) can only be either 0 or 1 and \( W(P_i, P_j) \) is positive are recognized. In the following, we prove property (4), the triangle inequality.

To prove the triangle inequality, it suffices to show that for any nonzero term in \( D_{W,K}(\pi, \sigma) \), the term also appears in \( D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma) \). In fact, if \( K(P_i, P_j) = 1 \) in \( D_{W,K}(\pi, \sigma) \), we know that the relative order between \( P_i \) and \( P_j \) are different in \( \pi \) and \( \sigma \). Without loss of generality, suppose \( \pi(P_i) < \pi(P_j) \) and \( \sigma(P_j) < \sigma(P_i) \). Because there are only two possible orders between \( P_i \) and \( P_j \) in a ranking, the relative order between \( P_i \) and \( P_j \) in \( \eta \) must disagree with that in either \( \pi \) or \( \sigma \). According to the definition of \( K(P_i, P_j) \), the term \( W(P_i, P_j) \) must also appear in \( D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma) \). Therefore,

\[
D_{W,K}(\pi, \sigma) \leq D_{W,K}(\pi, \eta) + D_{W,K}(\eta, \sigma) \quad (4.7)
\]

Meanwhile, if the relative order of \( P_i \) and \( P_j \) agrees between \( \pi \) and \( \sigma \), but disagrees with that in \( \eta \), the right side of Eq. 4.7 is larger. The equal sign holds when no such predicate pairs exist.

By Theorem 2, the weighted Kendall’s tau distance is a valid metric for distances between the projected rankings. So the distance between \( f_i \) and \( f_j \) is defined to be \( D_{W,K}(\tau_i', \tau_j') \), which equals to \( D_{W,K}(\tau_i, \tau_j) \). Continuing Example 2, we calculate \( D_{W,K}(\tau_1, \tau_2) = 0.07536 > 0.0625 = D_{W,K}(\tau_1, \tau_3) \), and this conforms to our expectation. Finally, we note that the weighted Kendall’s tau distance can be efficiently calculated. First, because counting the number of discordant pairs between two rankings is equivalent to a sort, the complexity of the classic Kendall’s tau distance is \( O(l \log(l)) \) [49], where \( l = |S_r| \). Second, the weights \( W(P_i, P_j) \)’s in Eq. 4.6 do not change the complexity. Therefore, the distance between two predicate rankings is calculated in \( O(l \log(l)) \) time. Experimental results about the time efficiency of R-PROXIMITY are provided in Section 4.5.3.
4.4 Advantages of R-Proximity

R-Proximity features (at least) three advantages over T-Proximity. They are fault-aware failure clustering, guided failure assignment and interpretation of debugging result. We discuss them in the following three subsections respectively.

4.4.1 Fault-aware Failure Clustering

R-Proximity is defined on the fault-aware fingerprints of failing traces, which are semantically closer to the underlying faults than literal execution traces are. Ideally, we want to partition failing cases properly so that failures due to the same fault are grouped together. However, since the “due-to” relationship between failing cases and underlying faults is unknown, the ideal partition is generally unachievable.

R-Proximity uses Sober to substitute for the manual investigation. In consequence, the fault location each failing case suggests is an approximation to the real “due-to” relationship. Therefore, under R-Proximity, failing traces whose induced fault localization agree with each other are regarded near to each other. In comparison, T-Proximity approximates the “due-to” relationship by hypothesizing that similar traces implies the same fault. As will be seen in Section 4.5, failing cases due to the same fault can actually exhibit quite divergent behaviors.

Although pair-wise distances suffice for failure clustering, it is usually beneficial to visualize the proximity between failing traces. In the first place, visualization is more intuitive for human beings to comprehend. In the second place, it makes it possible for human beings to visually identify failure clusters. Although many clustering algorithms have been developed so far [14, 21, 52, 11], visual inspection is still the most effective and accurate way to identify reasonable clusters.

In this article, we visualize the failure proximity through a statistical technique called multidimensional scaling (MDS) [15]. The MDS technique takes the pair-wise distances between \( n \) points, and tries to present them in a much lower dimensional (2-D here) space while preserving the original pair-wise distances. We call this 2-D visualization a proximity graph. When two points overlap in a proximity graph, it does not indicate that their original distance is zero. Instead, it only suggests that the distance between them is too small for MDS to visualize at the given scale. By visualizing the failure proximity in a 2-D graph, one can visually identify failure clusters, and
easily tell the differences between R-PROXIMITY and T-PROXIMITY.

4.4.2 Guided Failure Assignment

After failure clusters are identified, the member traces in each cluster need to be assigned to appropriate developers for patches. A developer is appropriate if she or he is responsible for the fault that fails most member cases in the cluster.

According to the definition of R-PROXIMITY, a failure cluster formed under R-PROXIMITY automatically provides a guess about the likely fault location for this cluster. Specifically, the fault location agreed by the member traces is the guessed location, and consequently, developers responsible for the guessed fault location should be assigned to investigate failures in this cluster. Therefore, under R-PROXIMITY, the manual investigation for failure assignment is saved.

Then how to find the agreement between member traces for each cluster? Let the individual ranking derived from a member case be a member ranking of the cluster. Because top-ranked predicates are regarded fault-relevant, predicates that are ranked high in most member rankings likely represent the agreed fault location. Therefore, a developer can skim over the top predicates of the member rankings in each cluster, and identify what predicates appear most often. These identified predicates represent the agreed fault location.

However, this manual approach is laborious, especially when hundreds of member traces are included in a cluster. We therefore propose a spectrum graph to facilitate the identification of agreed fault location. We define a predicate is favored by a failing trace if this predicate is ranked within the top-$k$ of the corresponding individual ranking. The parameter $k$ reflects one’s belief in what percentage of top percentages are really fault-relevant. Usually, $k = 3$ is an appropriate setting because rarely one would bother with lower ranked predicates.

Then we can count the top-$k$ favored frequency for each predicate, and plot the frequencies for all predicates. The top-$k$ favored frequency is the number of member traces that favor the predicate for a pre-set $k$. We call such a plot a spectrum graph because it resembles those used in spectrum analysis. In a spectrum graph, spikes correspond to the most favored predicates, and developers can readily identify them.

We expect that only a limited number of distinct spikes will appear in spectrum graphs. And
moreover, the set of spikes should be insensitive to the setting of $k$, as long as $k$ is reasonably small. The reason is as follows. First, the weighted Kendall’s tau distance is mainly determined by the top predicates of each ranking because of the predicate weighting schema. Second, for a reasonable $k$ there cannot be many distinct predicates. Otherwise, the mutual distances between member rankings would be large, and corresponding member traces would no longer cluster together under R-PROXIMITY. Readers can find concrete spectrum graphs and see how they function soon in the case studies in Section 4.5.

4.4.3 Interpretation of Debugging Results

Because the global debugging result $\tau$ is also a predicate ranking, nothing different from individual rankings, it can be plotted in proximity graphs with all individual rankings. This provides a means of assessing the quality of $\tau$ before really looking into it. We regard such an assessment useful because otherwise developers have no confidence in the quality of the debugging result. In fact, the lack of trustworthiness estimation is a common shortcoming for most, if not all, fault localization techniques, and here R-PROXIMITY provides an approach to alleviate this problem.

The assessment of the debugging result is based on whether $\tau$ is close to any individual rankings. As the first possibility, $\tau$ can be inside a failure cluster, or at least near to one or more individual rankings. In this case, we know the debugging result $\tau$ is explainable because some concrete failing traces do suggest the same fault location. In consequence, developers can pick up the nearest failing trace to $\tau$, and use it to understand and utilize the debugging result $\tau$. For example, developers can use the concrete failing trace to understand why certain predicates are ranked high in $\tau$. The case study with grep is prepared in Section 4.5.1 to illustrate this usage.

On the other hand, if $\tau$ is far way from all $\tau_i$’s, this means that no single failing trace can account for the debugging result. In consequence, developers may regard $\tau$ as inaccurate. Nevertheless, $\tau$ is automatically derived from runtime statistics with no prior knowledge about program semantics, and we cannot expect it is accurate all the time. If such situations are encountered, we can ignore $\tau$, and resort to other approaches. For example, we can re-run the debugging algorithm on each identified failure clusters separately. We will explain this situation with more details in Section 4.5.2, through the case study with gzip.
4.5 Advantages Illustrated with Case Studies

In this section, we report on two case studies to exemplify the advantages of R-PROXIMITY as discussed in the above section. They are presented in Sections 4.5.1 and 4.5.2, respectively. During the case studies, we also examine the time efficiency for computing R-PROXIMITY, and related results are reported in Section 4.5.3. Finally, Section 4.5.4 summarizes the conclusions drawn from the two case studies.

4.5.1 Case Study I: Grep-2.2

We obtained a copy of grep-2.2 and an accompanying test suite of 470 test cases from the “Subject Infrastructure Repository” (SIR) [32]. The grep program, as measured by the SLOCCount tool [93], has 15,633 lines of C code, excluding both blanks and comments. The source code is instrumented with 1732 boolean and 1404 return predicates, and three predicates that are referred to in the following discussion are listed in Table 4.1.

We manually injected two faults into the grep program, and they are shown in Figure 4.3. The first fault (Fault 1) is an “off-by-one” error: We added the “+1” at Line 553. This fault fails 48 of the 470 test cases. The second fault (Fault 2) is a “subclause-missing” error. We commented out the subclause \( lcp[i] == rcp[i] \) at Line 2270, and this incurs another 88 failing cases. We use \( F_1 \) and \( F_2 \) to denote the sets of failing cases due to Faults 1 and 2, respectively. When both faults are activated, 136 cases (denoted by \( F \)) fail, and \( F = F_1 \cup F_2 \).

Regarding the above description, one may wonder how the culpability is determined for each failing case in \( F \). Nevertheless, the execution of any failing case can be influenced by both faults, although it could actually fail due to a particular one. Ultimately, only experienced developers can determine which fault or faults are responsible for each failing case. But such a manual approach is usually too expensive to implement. Instead, we propose the following rule to determine the culpability for each failing trace. And the rule is that if a test case fails at the existence of multiple
faults, the culpability of the failing case is assigned to those faults which can fail the test case individually. For example, when there are two faults in a program, the culpability is determined by Table 4.2.

Table 4.2: Culpability Assignment at the Existence of Two Faults

<table>
<thead>
<tr>
<th>Situation</th>
<th>Fails or Not</th>
<th>Fault 1</th>
<th>Fault 2</th>
<th>Both</th>
<th>Culprit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Coexistence of Fault 1 and Fault 2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Fault 1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Fault 2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Both Fault 1 and Fault 2</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 determines the culprit by observing whether the test case fails when Fault 1 only, Fault 2 only or both faults are activated. For the eight possible combinations (or situations), the last column lists the convicted culprit. Because culprits are only convicted when failure is observed, there is no culprit for Situations 1 to 4. In Situation 5, the test case fails if and only if both faults...
are activated, and this suggests that it is the collaboration of both faults that fails the test case. In consequence, the culprit is the coexistence of both faults. For Situations 6 and 7, Fault 1 and Fault 2 are convicted as the culprit respectively, according to the aforementioned rule. Finally, both faults are the culprit for Situation 8, because the test case fails whenever either or both faults are activated.

Although eight situations are listed in Table 4.2, only cases in situation 1, 6 and 7 are observed in the case studies of grep and gzip. Situations 2 to 4 are not observed because they are intrinsically small-probability events. For example, in Situations 2 to 4, the existing failure disappears once both faults are activated, and this suggests that the fault is fixed by the other. Situation 5 is not observed either because it is also a small-probability event. The two faults need to collaborate to fail a test case. So, in general, we believe that it is typical not to observe Situations 2 to 5. On the other hand, although Situation 8 has a non-trivial probability to be observed, corresponding cases are still not encountered. Therefore, in the case studies of grep and gzip, one and only one fault is convicted as the culprit for every failing trace.

Without little difficulty, Table 4.2 can be generalized to scenarios when more than two faults exist, but it is beyond the scope of this article. Nevertheless, culpability assignment is only needed in controlled experiments, like the case studies here.

Finally, we note that although the two faults in grep, as well as the other two faults in gzip, are manually injected, they do mimic realistic programming errors. When developers are unclear about corner conditions, logic faults like “off-by-one” or “subclause-missing” usually sneak in. Logic faults generally do not crash programs, but only incur incorrect outputs. As a result, one cannot rely on crash scenes to categorize failing traces and fix the faults. Instead, the best that a developer can do is to randomly pick one failing case and manually trace its execution. With the grep subject program, for example, a developer needs to hunt for the faults in more than 15k lines of code. In the following three subsections, we illustrate how R-Proximity can help developers cluster failing traces and finally find the faults. Especially, we report on the case study result based on the claimed three advantages of R-Proximity over T-Proximity.
Fault-aware Failure Clustering

The proximity graphs for the 136 failing traces under R-PROXIMITY and T-PROXIMITY are presented in the left and the right subfigures in Figure 4.4 respectively. The middle subfigure is the close-up of the rectangular region labelled Cluster2. We use red crosses and blue circles to represent failing cases in $F_1$ and $F_2$, respectively, and the red cross bounded by a diamond symbol denotes the global ranking $\tau$. Ideally, blue circles are expected to cluster together, while being away from crosses.

As we can see from the left figure, the failing cases in $F_2$ do form a cluster under R-PROXIMITY, and the cluster is meanwhile away from most red crosses. By observing the left subfigure of Figure 4.4, one would identify the two clusters as shown. In comparison, the blue circles stretch in a line under T-PROXIMITY, and the clustering is not as dense as that under R-PROXIMITY. This indicates that failing cases due to the same fault can exhibit quite divergent behaviors, which explicitly undermines the hypothesis T-PROXIMITY relies on. But meanwhile, as SOBER is nonetheless a substitute for manual investigation, the fault “due-to” relationship is not exactly uncovered. In the left figure, a number of red crosses are near to the blue cluster, and the close-up observation shows that there is a red cross inside the blue cluster. This impurity is totally understandable because one cannot expect an automated tool to perform as accurately as human developers. Moreover, as will be seen in the following, this impurity is immaterial to the advantages of R-PROXIMITY.
Guided Failure Assignment

Besides providing denser failure clusterings, R-PROXIMITY also facilitates the assignment of member cases in each cluster to responsible developers. The cluster Cluster1 consists of 21 failing cases, all of which are in F1. In order to assign them to the appropriate developers, we need to check what predicates are most favored by the member cases. Figure 4.5 shows the top-k spectrum graphs for Cluster1, with k varying from 1 to 6. In each spectrum graph, the x-axis is for the predicate indices, and the y-axis shows how many member cases favor each predicate. In addition, the most favored predicate is marked in the figures with text.

As shown in the six subfigures, predicates $P_{1470}$ and $P_{1484}$ are clearly most favored by member cases in Cluster1. Moreover, as expected, when k increases from 1 to 6, no more predicates become comparably distinct. For example, no other predicates are favored by more than 5 cases even when $k$ is set to 6. This observation backs our reasoning on the insensitivity to the parameter $k$ (Section 4.4.2), and suggests that developers do not have to find a “golden” parameter to assign failing traces properly. Commonly, we recommend examining the spectrum graph for $k = 1$, 2, and 3, and this should be enough for human beings to figure out what predicates are most favored. Because predicates $P_{1470}$ and $P_{1484}$ are most favored, and they both point to the grep function, the 21 failing traces in Cluster1 are assigned to the developers responsible for the “grep.c” file. If needed, the assigned developers can be told that the function grep likely contains the fault.

Figure 4.5: Case Study with grep-2.2: Top-k Spectrum Graphs for Cluster1
A similar analysis is carried out for Cluster2, which consists of 24 failing traces from F1 and 88 failing traces from F2. Figure 4.6 shows the spectrum graphs derived from the member rankings in Cluster2, for k equals 1, 2 and 3. Clearly, P1952 is the most favored predicate, and no other predicates are comparably prominent. In consequence, the 112 failing cases in Cluster2 are assigned to the developers in charge of the “dfa.c” file. Similarly, if needed, the developers can be informed of the likely fault location. The developers can then pick up a failing case whose induced ranking has P1952 as the top, and trace it for debugging. As one may have noticed, the impurity in Cluster2 did not dilute the prominency of P1952 in the spectrum graphs, and the assignment of failing traces was not mislead.

There are three remaining traces outside Cluster1 and Cluster2, and they are not assigned because (1) they only account for a small percentage of all failing traces, and (2) not all failing traces need to be assigned at one time. When some faults are fixed through assigned traces, some unassigned cases may not fail any more. With R-PROXIMITY, we identify failure clusters of non-trivial sizes, and assign them to responsible developers in a guided way. In comparison, because clusterings under T-PROXIMITY are fault-unaware, developers need to manually investigate each failure cluster and assign failing traces accordingly.

**Interpretation of Debugging Results**

We now explain how R-PROXIMITY can help developers assess the debugging result, finally locate the two faults. We first run SOBER on the 136 failing and the 334 passing cases, and get the debugging result τ, whose top three predicates are P1470, P1484 and P1952. Details of the three predicates are in Table 4.1. As seen from the left subfigure of Figure 4.4, the debugging result τ is inside Cluster1, and a number of individual rankings are close to it. This indicates that for some
failing traces, each of them suggests the same fault location as \( \tau \). Therefore, the debugging result \( \tau \) is explainable because developers can trace one of these close failing traces and understand why top predicates in \( \tau \) are ranked high.

The closest individual ranking to \( \tau \) is easily found, and its corresponding failing case is the 10th failing case \( f_{10} \). In fact, \( \tau_{10} \) also ranks \( P_{1470} \) and \( P_{1484} \) as the top two predicates. We now use the concrete failing case to explain why \( P_{1470} \) and \( P_{1484} \) are ranked high.

In the execution of \( f_{10} \), the evaluation bias of predicate \( P_{1470} \) is 0.09. In contrast, the evaluation bias is 1 in 301 of the 334 passing traces, and it is 0.9286 in the other 33 passing traces. So it seems that the evaluation of \( P_{1470} \) is abnormally biased to false. Similarly, we find that the evaluation of \( P_{1484} \) is abnormally biased to true evaluations. By tracing the execution of \( f_{10} \), we find that normally the variable \( \text{beg} \) is expected to be equal to the variable \( \text{lastout} \) at line 574. However, with the “+1” added at line 553, \( \text{beg} \) is no longer equal to \( \text{lastout} \) for most cases. In consequence, the predicate \( P_{1484} \) tends to evaluate true, and this makes the variable \( \text{lastout} \) frequently reset to 0 at line 575. The reset of \( \text{lastout} \) finally causes the predicate \( P_{1470} \) at line 549 to evaluate as false. Therefore, based on the execution of \( f_{10} \), the localization result \( \tau \) is interpreted in a concrete way, which finally guides developers to fix the fault.

In this case, \( f_{10} \) is a proper failing case for manual debugging. However, such proper cases cannot be found under T-PROXIMITY, because there is no easy way to figure out whether a failing trace will suggest the same fault as \( \tau \). As an alternative, people may wonder what if a random failing case is selected. First, a random failing case may be due to a different fault, other than the one suggested by \( \tau \). In this example, one has a probability of 0.65 (88/136) to pick up a failing trace due to Fault 2, while the top predicates in \( \tau \) are actually about Fault 1. Furthermore, even if a failing trace due to Fault 1 is selected, the selected trace may also fail to explain \( \tau \). For example, there are 15 failing cases in \( F_1 \), in which the evaluation bias of \( P_{1470} \) is 1. These 15 cases exhibit no abnormal symptoms on \( P_{1470} \), and hence provide little help for debugging. In fact, the observation that not all failing cases are equally effective for debugging holds in general, and it well conforms to our debugging experience: In manual debugging, even for the same fault, some failing cases are easy to trace, whereas some others are hard.

After fixing the first fault, a second run of SOBER with the 88 failing and 382 passing cases puts
Figure 4.7: Proximity Graphs with R-Proximity (left) and with T-Proximity (right) for grep-2.2 $P_{1952}$ at the top. Figure 4.7 plots the proximity graphs for the 88 failing cases under R-PROXIMITY (left) and T-PROXIMITY (right) respectively. Similarly, with R-PROXIMITY, a proper failing case is easily found, and tracing it immediately clears the second fault.

4.5.2 Case Study II: Gzip-1.2.3

Table 4.3: Selected Predicates for gzip-1.2.3

<table>
<thead>
<tr>
<th>Ranks</th>
<th>Filename</th>
<th>Line Num.</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{1078}$</td>
<td>bits.c</td>
<td>165</td>
<td>$(\text{bi_valid} &gt; 8) == \text{true}$</td>
</tr>
<tr>
<td>$P_{1190}$</td>
<td>gzip.c</td>
<td>590</td>
<td>$(\text{force}) == \text{false}$</td>
</tr>
<tr>
<td>$P_{1210}$</td>
<td>gzip.c</td>
<td>607</td>
<td>$(\text{verbose}) == \text{true}$</td>
</tr>
<tr>
<td>$P_{1136}$</td>
<td>deflate.c</td>
<td>615</td>
<td>$(\text{match_length} &lt;= \text{max_insert_length}) == \text{true}$</td>
</tr>
<tr>
<td>$P_{1137}$</td>
<td>deflate.c</td>
<td>625</td>
<td>$(\text{match_length} != 0) == \text{true}$</td>
</tr>
</tbody>
</table>

Now we report on the second case study with gzip-1.2.3. The gzip program and the accompanying test suite of 217 test cases are also obtained from the “Subject Infrastructure Repository”. The gzip program has 6,184 lines of C and assembly code, and is instrumented with 808 boolean and 1071 return predicates. Some predicates that will be referred to later are presented in Table 4.3.

Two “subclause-missing” faults are injected into the gzip program, as shown in Figure 4.8. The two faults each fail 65 and 17 of the entire 217 test cases. Because gzip is an independent case study from grep, the two faults are still denoted by Fault 1 and Fault 2, respectively. Similarly, we use $F_1$ to refer to the 65 failing cases due to Fault 1, and $F_2$ for the 17 failing cases due to Fault 2. When both faults are activated, 82 test cases fail, which are exactly the union of $F_1$ and $F_2$.

According to the culpability assignment table (Table 4.2), Fault 1 and Fault 2 are the culprit for
Fault 1: A subclause missing error in deflate.c

Figure 4.8: Two Injected Faults in gzip-1.2.3

the failing traces in $F_1$ and $F_2$, respectively.

Figure 4.9: Proximity Graphs with R-Proximity (left) and T-Proximity (right) for gzip-1.2.3

The proximity between the 82 failing traces have been plotted under R-PROXIMITY (left) and T-PROXIMITY (right) in Figure 4.9. Red crosses represent the 65 failing cases in $F_1$, and blue circles represent the 17 cases in $F_2$. Without ambiguity, one would identify the two clusters as shown in the left subfigure. This clustering is nearly perfect because both clusters are pure, and are meanwhile separate from each other. In comparison, under T-PROXIMITY, there are two distinct subclusters of red crosses, and a blue circle is far from other circles. This again shows that failing traces due to the same fault can actually exhibit quite divergent behaviors.
After the two clusters are identified, appropriate developers can be found for each cluster by inspecting the spectrum graphs. Figures 4.10 and 4.11 present the spectrum graphs for Cluster1 and Cluster2, respectively, with $k$ varies from 1 to 3. Similar to the what was observed in the grep case study, only a limited number of predicates are favored in each cluster, and the set of favored predicates is insensitive to the setting of $k$.

Specifically, Figure 4.10 suggests that three predicates $P_{1078}$, $P_{1190}$ and $P_{1210}$ are most favored by the member traces in Cluster1. Because predicate $P_{1078}$ points to the function `bi_windup` and the other two predicates point to the function `treat_stdin`, the 65 failing cases in Cluster1 are assigned to developers in charge of the functions `treat_stdin` and `bi_windup`. Because the faulty function `deflate` connects the function call chain from `treat_stdin` to `bi_windup`, the assigned developers are appropriate.

The assignment of failing cases in Cluster2 is similarly straightforward. Figure 4.11 shows that all the 17 member traces rank predicate $P_{1136}$ at the top, and the predicate $P_{1137}$ is put as the second highest in 16 member rankings. The two predicates are too close to be distinguished in Figure 4.11(b). Because predicates $P_{1136}$ and $P_{1137}$ are the most favored predicates, failing traces in Cluster2 are assigned to developers in charge of the `deflate_fast` function, the exact faulty function.
Finally, let us examine whether the proximity graph under R-PROXIMITY can help developers find the two faults. Different from what was observed in the case study of grep, the left subfigure of Figure 4.9 shows that the debugging result $\tau$ is far from all individual rankings. This means that no single failing case can account for the debugging result, and consequently, the debugging result $\tau$ is regarded as less accurate. In fact, the top-3 predicates of $\tau$ all point to the function $\text{send}_\text{tree}$. Because we are unfamiliar with the code of gzip, and no single failing case can explain $\tau$, we failed to explain how the three predicates relate to the faults.

The reason why $\tau$ is far from all individual rankings is probably that Fault 1 and Fault 2 are semantically similar. Specifically, Fault 1 and Fault 2 are in functions $\text{deflate}$ and $\text{deflate}_\text{fast}$, respectively, and the two functions implement a similar task except the efficiency for certain situations. As a result, the abnormal behaviors due to Fault 1 and Fault 2 are intertwined together, and SOBER cannot separate the abnormal behaviors due to each fault. Finally, SOBER generates the debugging result $\tau$, which seems irrelevant to either fault. Such intertwined situations were not observed in the case study of grep, nor in previous studies [64, 59], because the faults there were semantically distant, and the abnormal behavior due to a particular fault outweighs that due to the other faults. This unsuccessful experience with SOBER, together with the above discussion, underlines the importance of properly clustering failing traces before fault analysis.

When the debugging result is found less accurate, i.e., being far from all individual rankings, there are still other options for developers to explore. For example, the developers can re-run SOBER on each identified failure clusters, and investigate the fault in each cluster. We applied SOBER to Cluster1 and Cluster2 separately, and accurate debugging results were obtained for both clusters. As another alternative, if member traces in a failure cluster have a high agreement about the fault location, i.e., densely clustered under R-PROXIMITY, one can choose a proper failing case to debug based on the individual rankings. For example, since the fault location is highly agreed within Cluster2, a developer can easily find a failing case that ranks $P_{1136}$ and $P_{1137}$ at the top, and start debugging. The principle is that the proximity graph under R-PROXIMITY visualizes the relationship between the debugging result and each individual failing traces, and one can utilize it in different ways.
4.5.3 Time Efficiency

Because SOBER needs to be invoked with every failing trace, people may wonder whether the computation of R-PROXIMITY will take quite a long time. In fact, since SOBER only carries numerical computation, it is light-weighted even when invoked multiple times. For example, in the case study with grep, it took 13.6 seconds to fingerprint the 136 failing traces due to the two faults (the first round), and 9.9 seconds for the 88 failing traces after Fault 1 is fixed (the second round).

Besides fingerprinting, the calculation of the pair-wise distance is also negligible. In the first round, predicate rankings are projected into a subspace of 33 predicates, and the calculation of the pair-wise distance with R-PROXIMITY and T-PROXIMITY takes 0.345 and 0.101 seconds, respectively. In the second round, the projected subspace has 26 predicates, and the time for R-PROXIMITY and T-PROXIMITY distance is 0.245 and 0.042 seconds. Because gzip is smaller than grep, and fewer predicates are instrumented in gzip, much less running time was consumed in the case study of gzip. All the above time is recorded on a 3.2GHz Intel Pentium-4 PC with 1GB physical memory, running Fedora Core 2.

4.5.4 Summary

In this section, we reported on two case studies to illustrate the three advantages of R-PROXIMITY as discussed in Section 4.4. In both cases, more meaningful clusters are observed with R-PROXIMITY than with T-PROXIMITY. Plus, we showed that the fault location each failure cluster suggests can be easily identified by virtue of the proposed spectrum graphs. Finally, we also illustrated the usage of R-PROXIMITY in assessing the quality of the statistical debugging result, and in selecting a proper failing trace to help developers understand the debugging result. However, as a software program can be extremely complex, and an unknown large number of faults can exist in it, we cannot guarantee good results with R-PROXIMITY at all time. Instead, the major objective of this section is to establish R-PROXIMITY as a new means of debugging aids, and to exemplify its unique advantages.
4.6 Parameter Effects on Failure Clustering

In Section 4.3.3, we introduced the weighted Kendall’s tau distance as the distance measure for the R-PROXIMITY. The weighted Kendall’s tau distance performs both predicate selection and predicate weighting simultaneously through the combined weighting schema Eq. 4.5. Here, we show that both predicate selection and predicate weighting are indeed necessary, and illustrate how the parameters in Eq. 4.5 could influence the clustering result. In particular, proximity graphs are used to visually demonstrate the effect of different parameters on clustering results.

The Siemens suite [44, 82] is used in this study. We examined the 130 faulty versions contained in the suite, and found that the parameter effect on the clustering result is clear on some faulty versions and meanwhile unclear on some others. There are a number of factors that could affect the clustering result. For example, if a faulty version has too few failing cases, like less than 30, the clustering changes are usually insignificant, and hence it is difficult to see how the parameters affect the clustering result. Besides the number of failing cases, the effectiveness of SOBER on any particular faulty versions is also a key factor. In our experiments, we did observe inaccurate fault analysis from SOBER on some faulty versions. Because it is unrealistic to present the clustering changes for all the 130 faulty versions in this article, we finally choose to report on the result with schedule V4, the fourth faulty version of the schedule program. The observed result is clear and conforms to the reasoning. We believe that the conclusions drawn about the parameter effects should hold in general.

The program schedule V4 is instrumented with 24 predicates, and the underlying fault fails 294 out of the 2650 test cases. For easy visual distinctions, every failing case is colored in the proximity graphs. Specifically, the color is determined by how far the suggested fault location is from the real fault location for each failing case. Because the $T$-score rightly measures this distance [64], the $T$-score is calculated from each individual ranking, and the corresponding failing case is colored according to the colorbar shown in Figure 4.12.
### 4.6.1 Is Predicate Selection Critical?

![Figure 4.13: Effect of Predicate Selection](image1.png)

In Eq. 4.5, by setting $\alpha = 0$ and $k_1 = L$, all predicates are selected. Figure 4.13 shows the clustering contrast when $k_1 = 24$ (i.e., when no predicates are excluded) versus when $k_1$ is of the default value 10. As we can see, no clear clustering of the failing cases exists in Figure 4.13(a). On the other hand, when the top-10 predicates of $\tau$ are selected, cases are well separated with consistent colors. Therefore, predicate selection is critical for R-PROXIMITY to cluster together failing cases suggesting similar faults.

### 4.6.2 Is Predicate Weighting Critical?

The frequency weighting implemented by Eq. 4.4 is also critical for proper groupings of failing cases under R-PROXIMITY. The intuition of frequency weighting is that predicates favored by more individual rankings should speak louder in deciding the distances between failing cases.

Specifically, we set $\alpha = 1$ to focus on the effect of frequency weighting in $W_2$, and Figure 4.14 shows the contrast. In Figure 4.14(b), the predicate weights are calculated by Eq. 4.4, whereas in Figure 4.14(a), the above calculated weights are wiped even. As we can see, without predicate

![Figure 4.14: Effect of Predicate Weighting](image2.png)
weighting, failing cases again are mixed together.

4.6.3 Parameter Effects

Now that both predicate selection and weighting are shown critical, we now examine how the clustering of failing cases evolves when parameters $k_1$ and $\alpha$ vary. With $\alpha$ set as 0, Figure 4.15 shows how the grouping of the failing cases of schedule V4 changes when $k_1$ varies from 5 to 11. When $k_1 = 5$, these cases are represented by nine points because different rankings become identical when they are projected into a small subspace. When $k_1$ increases, more details about the proximity between failing cases become available, and the clustering becomes stable at a certain stage, like $k_1 = 10$ in this case. Certainly, when $k_1$ reaches the maximum $L$, the clustering boundary is blurred, as shown in Figure 4.13(a).

We now examine how the balancing parameter $\alpha$ affects the grouping of failing cases. When $\alpha$ is small, the predicate weight is mainly determined by the global ranking $\tau$. In consequence, more points tend to cluster around $\tau$. Figure 4.16 shows how the clustering changes when $\alpha$ varies from 0 to 1. As one can notice, the diamond point, which denotes $\tau$, moves to the group edge from inside when $\alpha$ increases. Because one usage of R-PROXIMITY is to help find a failing case that explains $\tau$ for debugging, a small value of $\alpha$, like 0.1, is generally preferred.
4.7 Discussions

Here we discuss the related work and potential threats to validity.

4.7.1 Related Work

First, this work is closely related to statistical debugging and fault localization in general. Due to the high cost of manual debugging, numerous fault localization techniques have been developed recently, like Delta Debugging [95], its derivatives [23, 38, 72], NN [81], LIBLIT05 [59], SOBER [64] and TARANTULA [47]. This work relates to fault localization in the following ways. First, a statistical debugging tool (SOBER) is used in this study to fingerprint failing traces into predicate rankings, on which R-PROXIMITY is then defined. Conventionally, automated debugging tools are used for fault localization purpose only. To the best of our knowledge, this is the first piece of work that uses statistical debugging to fingerprint executions. We believe that the idea of representing failing traces by the fault they each suggest is compatible with other fault localization techniques, as long as a proper distance is defined between fault localization results. Second, we demonstrate that R-PROXIMITY can in turn help developers assess and interpret the statistical debugging result. Because the lack of trustworthiness estimation and interpretability are two common problems for most fault localization techniques, this study sheds some light on how the accessibility of automated debugging can be improved in the future.

Second, this study also relates to failure investigation and debugging aids. Dickinson et al. propose a technique called cluster filtering to assist developers in finding failing traces from a set of mostly passing executions [30, 31]. Later on, Podgurski et al. report a study on clustering failure reports [78]. In these studies, T-PROXIMITY is used to assess the execution similarity. In this article, we propose R-PROXIMITY as another definition of failure proximity, and demonstrate that R-PROXIMITY is more suitable for characterizing the semantic proximity between failures. Similar to previous work [30, 31, 78], multidimensional scaling (MDS) techniques are also used to visualize the failure proximity in this article. However, as the debugging result \( \tau \) can be visualized with all failing traces under R-PROXIMITY, our visualization also provides a convenient means to assessing and interpreting the debugging result. Previous work has seen a number of visual debugging aids [83, 74]. Although these aids do improve the accessibility of debugging results, they do not help
developers estimate the trustworthiness nor find a proper failing case to debug.

Finally, our work also relates to the analysis of rank data [68]. In practice, many kinds of data, especially those involving opinions and judgements like merchandize preferences and political elections, are represented as rank data. In this study, we fingerprint failing cases into predicate rankings, and this is the first time failing traces are represented as rank data. In consequence, some interesting questions can be explored. For example, in the future, we will investigate whether more accurate fault localization can be achieved by aggregating individual rankings. Furthermore, this work proposes a weighted form of the Kendall’s tau distance to accommodate the speciality of predicate rankings. We prove its validity as a distance metric, and demonstrate the critical role weighting plays in R-PROXIMITY.

4.7.2 Threats to Validity

There are a number of threats to the validity of the case study and experiments. First, although the four faults in grep-2.2 and gzip-1.2.3 mimic realistic “off-by-one” and “subclause-missing” errors, they are nevertheless injected by our authors. For this reason, more case studies on large programs with multiple real faults need to be performed in the future. In general, we expect that similar results will be observed because the effectiveness of R-PROXIMITY depends on the fault localization quality, and statistical debugging has been shown capable of locating real faults in large programs [58, 59, 64]. Nonetheless, more experiments are needed to prove or disprove this expectation. Second, hand-crafted test inputs, rather than operational traces collected from the field, are used in this study. In general, operational failing traces are more divergent from each other. As T-PROXIMITY relies on the literal trace similarity, more divergent traces will render T-PROXIMITY less effective in grouping together failing traces due to the same fault. In the case studies, we have observed that divergent traces can be fingerprinted into similar predicate rankings by SOBER, but it is yet unknown whether similar phenomena will be observed with operational traces. Finally, the case study with grep-2.3 illustrates how R-PROXIMITY helps developers understand and utilize the statistical debugging result, but the ultimate evaluation should be carried out with end-users. However, due to the difficulty (and expense) of controlled user studies, most fault localization related researches are currently evaluated by the authors [81, 58, 59, 64, 47, 98].
4.8 Conclusion

In this article, we proposed a new approach, called R-PROXIMITY, to assess the proximity between failing traces. We reason and experimentally validate that with R-PROXIMITY, failing traces due to the same fault tend to be grouped together, but not with T-PROXIMITY. In addition, R-PROXIMITY features some exclusive advantages over T-PROXIMITY. For example, we show that R-PROXIMITY can help developers assign failing traces to appropriate developers, and developers can assess and utilize the statistical debugging result by examining the failure proximity under R-PROXIMITY. A number of interesting topics merit further study. For example, it would be beneficial, as well as interesting, to examine what debugging algorithms besides SOBER can be used to fingerprint failing traces, and for each chosen algorithm, what distance metric is needed to properly measure the agreement between debugging results.
Chapter 5

Failure Indexing through Program Dynamic Slices

5.1 Introduction

Software end-users are the most powerful testers: They keep revealing software faults (i.e., bugs) in released software that has undergone rigorous in-house testing. In order to leverage end-users’ testing power, failure reporting components have been widely adopted in deployed software, with Microsoft Dr. Watson System [2] and the Mozilla Quality Feedback Agent [4] being the two most typical examples. When a program fails, the failure reporting component automatically collects relevant information of the failure, and (with the user’s permission) reports it to software vendors for failure diagnosis and patches. Recently, third-party libraries that implement such failure reporting functionalities have been released for both C++ and Java, so that any programs, disregarding their complexity, can have their own failure reporting channels. The authors have seen this in Google Toolbar and FreeCall, just to name a few.

The automatically collected failures reflect how the software is exercised in practice, and what software faults really bother the users. Therefore, an appropriate analysis of such failure repository will provide invaluable guidance for software maintenance and development. However, most utilities of such reported failures rely on the resolution of a critical problem: failure indexing, which asks how to identify all failures due to the same fault. If failure indexing can be nicely performed, most utilities of the collected failure data will become routine work. For example, some typical and important utilities are

- **Failure Prioritization**: Reported failures have different levels of severity, and the most severe failure should be diagnosed and fixed first. Typically, the severity of a failure is determined by how many reported failures are due to the same fault as this particular one. With the support of failure indexing, failures due to the same fault can be easily identified, and consequently the
diagnosis of failures can be prioritized.

- **Duplicate Failure Removal**: Because of the sheer number of reported failures, manual diagnosis of every failure is impractical. With the support of failure indexing, developers only need to diagnose one failure from each failure set that arises from the same fault.

- **Patch Suggestion**: When a new failure occurs, it can be easily checked whether this failure has been solved before through failure indices. If yes, the failure reporter can be automatically directed to the patch to resolve the problem.

Failure indexing can sometimes be straightforward, especially when apparently effective failure signature exists. A case in point is crashing failures, which manifest as program crashes. Usually, crashing failures are incurred by memory bugs, such as dereferences of NULL pointers and memory corruptions. For crashing failures, the crashing venue (e.g., the call stack trace at program crashes) is a great failure signature because failures from the same fault tend to (but not always) exhibit the same crashing venue. By virtue of the nearly one-to-one mapping relationship between crashing venues and faults, indexing of crashing failures has been very successful in practice, as evidenced by the success of the Microsoft Dr. Watson System.

However, in the case of noncrashing failures, failure indexing becomes elusive because no unanimous signature like a crashing venue for crashing failure exists. The reason is that noncrashing failures are mostly incurred by semantic bugs, which usually cause program malfunctions (e.g., incorrect outputs) without crashing the program. Since no apparently effective signature exists any more, how to index noncrashing failures becomes an interesting and challenging problem.

Previous studies propose two failure proximity measures, which can be used to index noncrashing failures. Podgurski et al. [78] propose the T-PROXIMITY, which assigns a small dissimilarity value to pairs of failures that exhibit similar execution traces. In consequence, under T-PROXIMITY, failures with similar behaviors (e.g., similar branching actions) are indexed together. Because T-PROXIMITY does not rely on the crashing venue, it can be used to index noncrashing failures. But one shortcoming of T-PROXIMITY is that failures due to different faults can exhibit quite similar behaviors (especially before faults are triggered), which renders T-PROXIMITY ineffective in discriminating failures due to different faults. Based on this observation, Liu and Han pro-
pose R-PROXIMITY, which extracts fault-relevant information from program failures, and indexes failures accordingly [63]. Because only fault-relevant information is considered, R-PROXIMITY is shown to be more effective than T-PROXIMITY in distinguishing failures due to different faults.

However, the effectiveness of R-PROXIMITY does not come for free. The fault-relevant information is extracted from each failure by contrasting the failure against a set of passing executions. Unfortunately, the availability of such a set of passing executions cannot be freely assumed in practice. In the first place, non-trivial overhead will be imposed on user sides if passing executions, in addition to failures, are collected from end-users. More importantly, users are very sensitive to privacy which could be potentially infringed by the collection of correct executions. This explains why only program failures are collected in practice. In general, the availability of a non-trivial set of passing executions cannot be assumed. Therefore, in this paper, we investigate how to index noncrashing failures as effectively as R-PROXIMITY but without assuming any passing executions.

We propose a dynamic program slicing-based approach to indexing noncrashing failures. Specifically, we take the backward slices from the program failure point as the failure signature, and quantify whether two failures are due to the same fault according to the similarity between their corresponding backward slices. For noncrashing failures, the failure point is the source code that generates the first erroneous output. The advantages of this dynamic slicing-based approach are as follows.

- In comparison with T-PROXIMITY, we use dynamic slicing techniques to exclude fault-irrelevant information that is otherwise considered by T-PROXIMITY. For the same reason as R-PROXIMITY, exclusion of the fault-irrelevant information will improve the effectiveness in indexing noncrashing failures.

- In comparison with R-PROXIMITY, the dynamic slicing-based approach completely eliminates the need of any passing executions, and hence can be used in practice where only program failures are collected.

We performed three case studies with grep, gzip, and flex, and the result demonstrated the above claim.

Although current practice only reports crashing failures from user sites, indexing noncrashing failures is not an unrealistic problem. A recent study of bug characteristics [56] shows that semantic
bugs have become dominant because of the wide adoption of excellent memory monitoring tools, such as Valgrind and Purify. Specifically, the authors find that semantic bugs account for 81.1-86.7% of the 364 bugs they examined, and the ratio is projected to increase as software matures. These semantic bugs mainly manifest as wrong outputs, performance degradation, and incorrect functionality, which are all noncrashing failures. More importantly, the authors find that 71.9-83.9% of security bugs are also semantic bugs, and security break-ins always take place without crashing the program. Because of the increasing dominance of semantic bugs and the resulting noncrashing failures, we believe that the collection of noncrashing failures will be supported in the near future. Because no unanimous indexing techniques exist for indexing noncrashing failures, a systematic study of existing ones and investigation of new indexing techniques are in great need.

In summary, we make the following contributions in this paper.

- We pose the problem of indexing noncrashing failures, an increasingly critical problem due to the dominance of semantic bugs in the future.

- We propose a distance metric-based framework, which incorporates existing approaches and our proposed one. In order to foster future developments, a quantitative measure of indexing effectiveness is proposed within this framework, so that future techniques can be objectively evaluated.

- We propose a dynamic slicing-based approach to indexing noncrashing failures, which are advantageous over existing techniques. Three case studies on non-trivial programs are performed, and the result clearly validates our claim. To the best of our knowledge, this is the first attempt of using dynamic slices in failure indexing.

The rest of this paper is organized as follows. Section 5.2 explains the distance metric-based framework for failure indexing, and Section 5.3 discusses our dynamic slicing-based approach with references to the framework. We report the experiment results in Section 5.4. The related work and threats to validity are discussed in Section 5.5, and Section 5.6 concludes this study.
5.2 A Distance Metric-based Framework for Failure Indexing

Intuitively, failure indexing tries to compute a failure signature (i.e., the index) for each program failure, such that failures due to the same fault can be identified through the similarity between failure signatures. While this explanation suffices for intuitive understanding, a precise formulation facilitates unambiguous discussion and potentially fosters healthy development in the future. Therefore, in this section, we present a distance metric-based framework for failure indexing, which incorporates both existing approaches and our proposed one.

5.2.1 Failure Indexing in Formulation

Suppose a set of $n$ failures $X = \{x_1, x_2, \cdots, x_n\}$ is collected from a program $P$, and the $n$ failures are due to $m$ (unknown) faults $F = \{f_1, f_2, \cdots, f_m\}$. An oracle function $\Phi$, which is also unknown, specifies the due to relationship between $X$ and $F$, namely,

$$\Phi(x) = k \iff \text{the failure } x \text{ is due to fault } f_k,$$

and the fault $f_k$ is the root cause of the failure $x$. For clarity, we only consider failures that are induced by one fault at runtime even though multiple faults may reside in the program.

The oracle function $\Phi$ partitions the set of failures $X$ into $m$ mutually exclusive and collectively exhaustive sets:

$$S_k = \{x_i | \Phi(x_i) = k, \text{ for } i = 1, 2, \cdots, n\}.$$

For any failure $x_i$, $G(x_i)$ is the failure group that $x_i$ belongs to, and $G(x_i)$ includes all the failures due to the same fault as $x_i$, namely,

$$G(x_i) = \{x_j | \Phi(x_j) = \Phi(x_i), \text{ for } j = 1, 2, \cdots, n\},$$

and $x_i$ is a member of $G(x_i)$. With the above definitions, we can formulate failure indexing within a distance metric-based framework as below.

A failure indexing technique is a function pair $(\phi, D)$, where the function $\phi$ is a signature function, and the function $D$ is a distance function that is defined on a pair of signatures returned
Table 5.1: Different Indexing Techniques under the same Distance Metric-based Framework

<table>
<thead>
<tr>
<th></th>
<th>$\phi(x)$</th>
<th>$D(\phi(x_i), \phi(x_j))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>The optimal index</td>
<td>$\Phi(x_i)$, i.e., the root cause of $x_i$</td>
<td>1 if different root causes and 0 otherwise</td>
</tr>
<tr>
<td>T-PROXIMITY</td>
<td>Profile of the whole execution</td>
<td>Euclidean distance and city-block distance</td>
</tr>
<tr>
<td>R-PROXIMITY</td>
<td>A ranking of fault-relevant predicates</td>
<td>Weighted Kendall’s tau distance</td>
</tr>
<tr>
<td>Dynamic slice index</td>
<td>Dynamic slices from failure point</td>
<td>Set-based distance</td>
</tr>
</tbody>
</table>

by $\phi$. Specifically, function $\phi$ takes a program failure $x$ as input, and returns a failure signature; the distance function $D$ quantifies how failures are close to each other based on the similarity between their corresponding failure signatures. Usually, we require the distance function $D$ be a metric, meaning that the following four properties are satisfied:

1. $D(\alpha, \beta) \geq 0$ (non-negativity),
2. $D(\alpha, \beta) = 0$ iff $\alpha = \beta$ (identity),
3. $D(\alpha, \beta) = D(\beta, \alpha)$ (symmetry),
4. $D(\alpha, \gamma) \leq D(\alpha, \beta) + D(\beta, \gamma)$ (triangle inequality),

where $\alpha$, $\beta$, and $\gamma$ are three failure signatures.

Then a pair-wise distance matrix $M_{(\phi,D)}$, which is called the proximity matrix, can be calculated for the given set of $n$ failures, where

$$M_{(\phi,D)}(i, j) = D(\phi(x_i), \phi(x_j)).$$

A small value of $M_{(\phi,D)}(i, j)$ means that failures $x_i$ and $x_j$ are similar, and are likely to be indexed together by the indexing technique $(\phi, D)$. Each indexing technique defines a failure proximity, which is embodied by the proximity matrix.

Within this framework, the optimal indexing technique $(\phi, D)$ will minimize the intra-group distances,

$$\min_{\Phi(i)=\Phi(j)} \sum M_{(\phi,D)}(i, j),$$

and meanwhile maximize the inter-group distances,

$$\max_{\Phi(i)\neq\Phi(j)} \sum M_{(\phi,D)}(i, j).$$
Certainly, distances defined on different failure signatures must be first normalized before comparison. We will discuss a normalized measure in Section 5.2.2.

Previous studies, as well as the optimal indexing and our dynamic slicing-based approach, all fit into this distance metric-based framework, and Table 5.1 lists what functions are actually used in different indexing techniques. Especially, the first row of Table 5.1 indicates that if the oracle function $\Phi$ were known, the optimal indexing becomes a routine work. Because $\Phi$ can only be obtained through expensive manual work, our objective is to investigate automated indexing techniques that approximate the optimal one. In the next subsection, we propose an evaluation metric that quantifies the effectiveness of each indexing technique.

### 5.2.2 An Evaluation Metric

An evaluation metric should be independent of how indexing techniques are implemented, i.e., it does not need to know what $\phi$ and $D$ are; instead, the evaluation metric should only care about the proximity matrices that are generated by different indexing techniques. Besides the independence of indexing details, a good metric needs to consider the following two aspects:

- **Cohesion**: To what extent failures in the same group are close to each other;
- **Separation**: To what extent failures in different groups are separated from each other.

An excellent indexing technique will generate a proximity matrix that exhibits both high cohesion and high separation. In order to consider both cohesion and separation simultaneously, we propose the following metric, which borrows the idea of the Silhouette coefficient (SC) [85]. The Silhouette coefficient was originally proposed to evaluate the internal structure of data clustering results without knowing what data should be clustered together. Here, as we do know what failures should be indexed together, the Silhouette coefficient can be adapted to evaluate how effective an indexing technique is.

Specifically, the Silhouette coefficient (SC) of each failure $x_i$ is defined as

$$SC(x_i) = \frac{b_i - a_i}{\max\{a_i, b_i\}}$$  \hspace{1cm} (5.1)
where

\[ a_i = \frac{\sum_{x_j \in G(x_i)} M(i,j)}{|G(x_i)|} \]

and

\[ b_i = \min_{k=1,2,...,m,k \neq \Phi(x_i)} \frac{\sum_{x_j \in S_k} M(i,j)}{|S_k|}. \]

Intuitively, \( a_i \) is the average distance from \( x_i \) to all other failures in the same group. To compute \( b_i \), we first calculate the average distances between \( x_i \) and failures in \( S_k \) for all \( k \neq \Phi(x_i) \), and \( b_i \) is the minimum value among the \( m - 1 \) average distances.

Apparently, \( SC(x_i) \) varies between -1 and +1. A negative value is undesirable because it suggests \( x_i \) is closer to a group it does not belong to than to its own group. On the other hand, a positive value means \( x_i \) is close to other failures in the same group. After getting the Silhouette coefficients of each failure, the overall Silhouette coefficient, calculated from a proximity matrix \( M \), is

\[ SC(M) = \frac{\sum_{i=1}^{n} SC(x_i)}{n}. \quad (5.2) \]

Again \( SC(M) \) ranges from -1 to 1, and a high value indicates that the indexing technique \((\phi, D)\) is effective in indexing the given \( n \) failures. It is easy to verify that \( SC(M) \) is 1 for the optimal indexing technique. We illustrate the meaning of Silhouette coefficient through the following example, and conclude this section.

**Example 3** Suppose we have six failures \( \{x_1^6\}_{i=1} \) that are due to two faults. Explicitly, \( \Phi(x_1) = \Phi(x_2) = \Phi(x_4) = 1 \) and \( \Phi(x_3) = \Phi(x_5) = \Phi(x_6) = 2 \). In Figure 5.1, we use red crosses to represent failures in failure group \( S_1 \), and blue circles for failures in \( S_2 \). Figure 5.1(a) visualizes the optimal indexing, and Figure 5.1(b) plots a sub-optimal indexing case, where \( x_2 \) deviates from its group members.

For the sub-optimal indexing case, according to Equations 5.1 and 5.2, \( a_2 = 4, b_2 = 1, \) and \( SC(x_2) = -0.75 \), which reflects \( x_2 \)'s deviation from its failure group. Similarly, we can calculate \( SC(x_1) = SC(x_4) = 0.6 \) and \( SC(x_3) = SC(x_5) = SC(x_6) = 1 \), and the overall Silhouette coefficient for this indexing is 0.575.
In this section, we discuss the dynamic slicing-based approach to noncrashing failure indexing. Specifically, Section 5.3.1 discusses dynamic slicing techniques that serve as the signature function $\phi$, and Section 5.3.2 explains the distance function $D$ defined on dynamic slices. Finally, in Section 5.3.3, we describe a technique that visualizes failure indexing result.

### 5.3.1 Dynamic Slices as Failure Signatures

Dynamic slicing, invented as a debugging aid [51], is able to identify a subset of program statements that are involved in producing a program failure. Dynamic slicing operates by observing the execution of the program on a given input and collecting the dependences between executed statements. These dependences are used to compute dynamic slices.

Because a statement $s$ can be executed multiple times for a given input, we distinguish different execution of the same statement $s$ by *execution instances*. Suppose $s$ is executed $n$ times, we use $s_1, s_2, \cdots, s_n$ to denote the $n$ execution instances.

A dynamic slice is computed w.r.t. a specific execution instance $s_i$. In this paper, as we will use dynamic slicing techniques as the signature function $\phi$, dynamic slices are computed w.r.t. program *failure points*. For noncrashing failures, the failure point is the statement instance that produces
the first erroneous output. We now describe different types of dynamic slices that are used in this study.

**Data Slice (DS).** Statements that directly or indirectly influence computation of the faulty output through chains of *dynamic data dependences* are included in data slices. Formal definitions are as follows.

**Definition 4 (Dynamic Data Dependence)** An execution instance $s_i$ of the basic statement $s$ has a data dependence on the execution instance $t_j$ of the statement $t$, denoted as $s_i \xrightarrow{dd} t_j$, if and only if there exists a variable $\text{var}$ whose value is defined at $t_j$ and is then used at $s_i$.

**Definition 5 (Data Slice)** The data slice of an execution instance $s_i$, denoted as $DS(s_i)$, is

$$ DS(s_i) = \{s\} \cup \bigcup_{t_j, s_i \xrightarrow{dd} t_j} DS(t_j). $$

Figure 5.2 (left) shows an example of DS. It presents an execution trace instead of the static source code even though the code is self-explicit from the trace. This is also the case in the rest of the paper unless otherwise specified. In this example, there are data dependences between 30 and 40, and between 10 and 30. Therefore, the data slice of the value $z$ at 40 includes 10, 30, and 40.

Note that even though dependences are defined between statement *instances*, a slice contains unique statements instead of statement instances. In other words, a statement appears in the slice only once even when multiple instances of the statement are involved in computation of the faulty value.

**Full Slice (FS).** Statements that directly or indirectly influence the computation of faulty
output value through chains of dynamic data and/or control dependences are included in full slices [51].

**Definition 6 (Dynamic Control Dependence)** A statement execution instance \( s_i \) of statement \( s \) has a control dependence on the execution instance \( t_j \) of statement \( t \), denoted as \( s_i \xrightarrow{cd} t_j \), if and only if

1. statement \( t \) is a predicate statement, and
2. the execution of \( s_i \) is the result of the branch outcome of \( t_j \).

**Definition 7 (Full Slice)** The full slice of an execution instance \( s_i \), denoted as \( FS(s_i) \), is

\[
FS(s_i) = \{ s \} \cup \bigcup_{\forall t_j, s_i \xrightarrow{cd} t_j \text{ or } s_i \xrightarrow{dd} t_j} FS(t_j).
\]

Figure 5.2 (right) shows an example of FS. The control dependence \( 31_1 \xrightarrow{cd} 30_1 \) renders both statements 30 and then 20 included in the full slice.

**Pruned Slice (PS).** In [97], a technique is proposed to compute for each executed statement the likelihood of it being faulty. The basic idea is derived from the observation that some of the statements used in computing an incorrect value may also have been involved in computing correct values. Hence, it is possible to estimate the likelihood of a statement being faulty by checking its relationship to both correct and incorrect outputs. The estimated likelihood is called confidence value, which ranges from 0 to 1: A higher value means the statement is more likely to produce correct outputs. In consequence, statements with a higher confidence value can be pruned from the original slice, which can be either a data slice or a full slice.

Figure 5.3 gives an example of pruned slice. Suppose a user observes that statement instance \( 30_1 \) outputs a correct value while \( 31_1 \) outputs a wrong one, so that they are associated with the confidence values of 1 and 0, respectively. The definition at statement instance \( 22_1 \) only reaches the wrong output point, which can be interpreted as no evidence in \( 22_1 \) producing a correct value. Therefore, \( 22_1 \) has a confidence value of 0. The definition at \( 21_1 \) transitively reaches both the correct and the wrong outputs. From the fact that \( y \) at \( 30_1 \) is observed to be correct, we can infer \( 20_1 \) produces a correct value and hence its confidence value is 1. Similarly, we infer that
10. \( x=1; \quad C = f(\text{range}(x)) \quad ? \)

20. \( \text{if } (x<2) \quad C = 1 \quad \checkmark \)
21. \( y=... \quad C = 1 \quad \checkmark \)
22. \( z=x...y.. \quad C = 0 \quad \times \)

30. \( \text{print}(y); \quad C = 1 \quad \checkmark \)
31. \( \text{print}(z); \quad C = 0 \quad \times \)

\[ \text{PS}(31,30) = \{10, 22, 31\} \]

Figure 5.3: Pruned Slice (PS)

20 has confidence of 1. However, from 20 being correct, we cannot infer 10 is correct because the computation at 20 represents a many-to-one mapping, namely, from an integer domain to a boolean domain. Therefore, the confidence of 10 is computed based on the range of \( x \), which can be approximated by value profiles. Finally, all instances that have a confidence value of 1 are eliminated from the original slice. Therefore, \( \text{PS}(31,30) = \{10, 22, 31\} \), where 31 and 30 stand for the incorrect and correct output points. Depending whether the pruning is performed on full slices or data slices, the resulting pruned slices are called pruned full slices (PFS) or pruned data slice (PDS). Readers interested in more details about pruned slices are referred to [97].

### 5.3.2 Distances between Dynamic Slices

By taking dynamic slicing as the signature function \( \phi \), each failure is represented by a dynamic slice. Therefore, an appropriate distance function \( D \) that is defined on dynamic slices is needed to complete the dynamic slicing-based failure indexing. Given that a dynamic slice is essentially a set of statements, any distance metric defined on sets suffices. In this study, we choose the Jaccard distance, which was originally proposed by Levandowsky and Winter [54].

**Definition 8 (Distance between Dynamic Slices)** For any two non-empty dynamic slices \( e_i \) and \( e_j \) of the same program \( P \), the distance between them is

\[
D(e_i, e_j) = 1 - \frac{|e_i \cap e_j|}{|e_i \cup e_j|}.
\]

This distance is a valid metric. Readers interested in the proof of the triangle inequality are referred to [54].
The distance $D$ completes our dynamic slicing-based approach to failing indexing. Depending on what dynamic slices are chosen as failure signatures, we have a series of four indexing techniques: FS-PROXIMITY, DS-PROXIMITY, PFS-PROXIMITY and PDS-PROXIMITY, whose meanings are self-explained.

### 5.3.3 Failure Indexing in Visualization

The Silhouette coefficient discussed in Section 5.2.2 numerically summarizes the effectiveness of an indexing technique; consequently, different indexing techniques can be quantitatively compared. However, the ultimate goal of failure indexing is not to compare different techniques, but rather to help developers explore a (potentially huge) set of failures. A typical task of failure exploration is to identify the largest subset of failures that are likely due to the same fault for the purpose of failure prioritization. For this reason, we believe that a frontend that visualizes the indexing result of a set of failures will greatly assist users’ failure exploration. In addition, the visualization also provides us with an intuitive approach to comparing different indexing techniques, i.e., we can visually assess the cohesion and separation of a given indexing result.

For the same reason as the Silhouette coefficient, the visualization should only rely on the proximity matrix $M$. The dependence on neither original failure data nor failure signatures makes it compatible with any distance metric-based failure indexing techniques to be developed in the future. For this reason, we choose to use the multi-dimensional scaling (MDS) techniques [15], which visualize the proximity between the $n$ failures given a proximity matrix $M$.

The obstacle that MDS techniques want to overcome is that the $n$ objects whose pair-wise distances are specified by $M$ could originally reside in a very high-dimensional space. For example, in our case, each failure is in a space of hundreds of dimensions because a typical slice contains hundreds of statements. Apparently, we cannot visualize the proximity between the $n$ failures in the original space. Instead, what we can do is to re-arrange them in a specific way in a much

| Program | SLOC | Test # | Fault 1     | Fault 2          | Failure # | $|S_1|$, | $|S_2|$, | $|S_1 \cap S_2|$ |
|---------|------|--------|-------------|-----------------|-----------|------|-------|-----------------|
| grep-2.2 | 15,633 | 470   | Off-by-one  | Subclause-missing | 136       | 48   | 88    | 0               |
| gzip-1.2.3 | 6,184  | 217   | Subclause-missing | Subclause-missing | 82        | 65   | 17    | 0               |
| flex-2.4.7 | 9,212  | 525   | Off-by-one  | Off-by-one      | 255       | 163  | 92    | 0               |

- **Table 5.2: Characteristics of the Three Subject Programs**
lower (usually 2) dimensional space such that the pair-wise distances are best preserved. Readers interested in the technical details of MDS are referred to [15].

We call the visualization of an indexing result a proximity graph. Since the only objective of MDS techniques is to best preserve the original distances in a much lower dimensional space, the axes in a proximity graph are meaningless. A caveat that one should keep in mind while interpreting a proximity graph is that the proximity graph is not a projection of the original data into a low-dimensional subspace. Explicitly, a large distance between two objects in a proximity graph just indicates that the two objects are far from each other in the original space. No projection should be applied to proximity graphs.

5.4 Experiment Result

In this section, we report experiment results that demonstrate the effectiveness of dynamic slicing-based indexing techniques. We compare FS-PROXIMITY and DS-PROXIMITY with T-PROXIMITY and R-PROXIMITY through three case studies in Section 5.4.2, and explore the indexing effectiveness of pruned slices in Section 5.4.3. Before going into details about experiment result, we first describe our experiment setup in Section 5.4.1.

5.4.1 Experiment Setup

We obtained three subject programs, gzip, grep, and flex, together with the accompanying test suites from the “Software-artifact Infrastructure Repository” (SIR) [32], which “is a repository of software-related artifact meant to support rigorous controlled experimentation.” The exact version number, the number of physical Source Lines of Code (SLOC), and how many tests are included in the test suite are listed in Table 5.2. Especially, the program size (SLOC) is measured by the SLOCCount Tool\(^1\).

Six semantic bugs are seeded into the three subject programs as depicted by the 5th and 6th columns of Table 5.2. We will discuss the faults in detail in each case study. The 7th column of Table 5.2 lists how many test cases fail because of the two seeded faults for each subject. In particular, all failures manifest as incorrect outputs without crashes, and are hence noncrashing.

\(^1\)http://www.dwheeler.com/sloccount/
The system we used to collect dynamic slices is shown in Figure 5.4, which consists of the static and the dynamic components. The static component computes static control dependence that is required to instrument the program binary. The static analysis was implemented using the Diablo [3] retargetable link-time binary rewriting framework. Diablo has the capability of constructing the control flow graph from x86 binary. The static control dependence information is indexed by the virtual addresses such that it can be shared by both components. The dynamic component of the system, which is based upon the Valgrind [84], accepts the same binary and dynamically instruments it by calling the slicing instrumenter. The instrumented code is executed with the support of the slicing runtime. The slicing instrumenter and slicing runtime were developed to enable collection of dynamic dependence information, which is later used in computation of dynamic slices. Valgrind can be considered as a dynamic instrumentation engine. During program execution, each new (never instrumented) basic block is instrumented by calling the instrumentation function provided by the slicing instrumenter. The kernel executes the instrumented basic block instead of the original one. The instrumented basic block is copied to a new code space and thus it can be reused without calling the instrumenter again. The slicing runtime essentially consists of a set.
of call back functions for certain events (e.g., entering functions, accessing memory, arithematic operations, and predicates).

Table 5.3: Failure Group Determination of Failure $x$

<table>
<thead>
<tr>
<th>Situation</th>
<th>Fails or Not</th>
<th>Failure Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

For evaluation purpose, we need to determine the failure group for each program failure. Precisely, one needs to manually investigate each failure $x$, and decides whether $x \in S_1$ or $x \in S_2$, or even both for some extreme cases. However, examining 473 (136+82+255) failures entails too much manual effort; instead, we determine the failure group for each failure through the following procedure, which we believe accurately determines the true failure group membership for each failure.

For each subject program, we first activate both faults and run the faulty program through the whole test suite. The number of failed test cases (i.e., failures) is listed in the 7th column of Table 5.2. This is the set of failures we want to index. Then we run through the test suite with one and only one fault activated each time, and the 8th and 9th columns of Table 5.2 list how many cases fail due to each fault, respectively.

The four interesting situations are presented in Table 5.3, which determines the failure group for each failure. In our experiment, all failures fall into Situations 2 and 3, as indicated by the last column of Table 5.2.

As one may have noticed, we do not consider scenarios with more than two faults. We focus our discussion on the two-fault scenario because (1) the purpose of this study is to compare the dynamic slicing-based approach with existing techniques, and (2) we believe that no fundamental difference exists between two-fault and three-fault for the purpose of studying indexing effectiveness. Therefore, we restrict our case studies to the two-fault scenarios, and leave more-fault scenarios to future studies.
Figure 5.5: Comparison with T-Proximity and R-Proximity on Grep-2.2

5.4.2 Comparison with T- and R-Proximity

In this section, we compare FS-PROXIMITY and DS-PROXIMITY with T-PROXIMITY and R-PROXIMITY through three case studies. We leave the exploration of PFS-PROXIMITY and PDS-PROXIMITY to Section 5.4.3 because pruned slices to some extent utilize information about program correct executions.

Case Study 1: Grep-2.2

The two seeded faults in grep are the same as the two used with R-PROXIMITY in [63], so the code of the two faults is not reproduced here because of space limit. Figure 5.5 shows the proximity graphs for the four indexing techniques. The Silhouette coefficients are annotated at the bottom of each graph in parenthesis. We represent failures in $S_1$ and $S_2$ by red crosses and blue circles,
respectively. In proximity graphs, better indexing effectiveness is indicated by higher brightness of the same color, which is best viewed with color printing.

Figure 5.5(a) suggests that failure indexing with T-PROXIMITY is not good. Especially, the line of blue circles indicates that failures in \( S_2 \) are not well indexed by T-PROXIMITY. Although the strong linear pattern of blue circles tends to suggest an appropriate projection will coalesce all blue circles, such projection is invalid because of the reason discussed at the end of Section 5.3.3. In contrast, since the pair-wise distances between failures in \( S_2 \) are small under R-PROXIMITY, all blue circles are densely clustered together (Figure 5.5(b)). Similarly, blue circles are also close to each other with both FS-PROXIMITY and DS-PROXIMITY.

On the other hand, the scattered red crosses indicate that no indexing techniques can index failures in \( S_1 \) well. In comparison, the result with DS-PROXIMITY appears to be the best among the four, because only three failures in \( S_1 \) are around the blue cluster, and all other red crosses form two clean clusters, which are not far apart from each other. This explains why DS-PROXIMITY has the highest Silhouette coefficient in this case study.

Fault 1: A subclause missing error in deflate.c

Fault 2: Another subclause missing error in deflate.c

Figure 5.6: Two Seeded Faults in Gzip-1.2.3
Case Study 2: Gzip-1.2.3

Two “subclause-missing” errors are seeded into gzip, which are depicted in Figures 5.6. We manually check the 217 failures from the two faults, and find all failures have the same failure point. This suggests that indexing by the failure point, which is the simplest slice, is not effective.

Figure 5.7 plots the proximity graphs for the four indexing techniques. Interestingly, we notice that the deviating blue circle in Figure 5.7(a) moves closer and closer to the blue cluster with R-PROXIMITY (Figure 5.7(b)) and FS-PROXIMITY (Figure 5.7(c)), and finally completely merges into the cluster with DS-PROXIMITY. This suggests that some failures that are not correctly identified by T-PROXIMITY can be correctly indexed by dynamic slicing-based approaches. In addition, DS-PROXIMITY has also done a great job in indexing failures in $S_1$: The red crosses clearly form two cohesive and dense clusters in Figure 5.7(d). This is a very nice property because
a duplicate failure remover will have a high confidence in keeping just one representative failure from each dense cluster and throwing away the rest.

Although DS-PROXIMITY appears to achieve the best indexing result in Figure 5.7, its Silhouette coefficient is strangely low. Apparently, the low coefficient comes from the large distance between the two red clusters. Then, a natural wonder is that given that all red crosses represent failures in $S_1$, why are they separated into two clusters?

We manually investigate the two red clusters in Figure 5.7(d), and find that the two clusters correspond to two different failing mechanism although they are all due to Fault 1 (Figure 5.6). We select a representative from each cluster (test cases 8 and 82 respectively), and explain how they fail differently from the same fault. Because slices in the same cluster are nearly identical, it does not matter which particular failure is chosen.

Figure 5.8 presents the data slice of case 8. The wrong value is observed at statement 134 in function `send_bits()`, which is called and passed with a faulty parameter at line 1033. The faulty parameter is produced by the data dependence chain of $1033 \rightarrow 1032 \rightarrow 1026$. A further study of Fault 1 in Figure 5.6 reveals that the faulty branch at statement 686 produces a faulty `match_length`, which makes the control flow select the wrong branch at 707. This in turn results in `ct_tally()` being called by mistake at line 738. Inside this call, the array `l_buf` is polluted. Finally, when the execution tries to print a compressed block that is affected by `l_buf`,
an incorrect output is observed.

Figure 5.9 presents the data slice of case 82. In this failing case, the wrong output is observed at the same source code location (statement 134) as case 8. However, the failure follows a completely different dependence path. At the function level, the dependence chain is
\[
\text{send_bits} \xrightarrow{dd} \text{send_tree} \xrightarrow{dd} \text{gen_codes} \xrightarrow{dd} \text{gen_bitlen} \xrightarrow{dd} \text{pqdownheap} \xrightarrow{dd} \text{ct_tally}.
\]

The explanation is that \texttt{ct\_tally()} is mistakenly called at line 707 due to Fault 1. The function \texttt{ct\_tally()} calculates the frequencies of different trees, which are used to encode bytes in \texttt{gzip}. Because of Fault 1, the faulty frequency calculated by \texttt{ct\_tally()} results in wrong trees being constructed, which are eventually dumped to the output by the function \texttt{send\_tree()}, as part of the entire output.

Therefore, the case study with \texttt{gzip} clearly indicates that the same fault can fail the program
750 void readin()
775   if ( performance_report >= 0 ) /* Fault1: should be > */
777     if ( lex_compat )
785     if ( performance_report > 1 )
796     if ( reject )
800   if ( variable_trailing_context_rules )

Fault 1: an off-by-one error in main.c
Fault 2: an off-by-one error in gen.c

Figure 5.10: Two Seeded Faults in Flex-2.4.7

... in totally different ways, and that DS-PROXIMITY explicitly indexes failures with different failing mechanism apart. While this is intuitively an advantage, DS-PROXIMITY is nevertheless penalized by the Silhouette coefficient for it. This raises our wonder about whether the optimal indexing should index all failures due to the same fault together, or should only index failures with similar failure mechanism together. For some applications, like failure prioritization, the former is preferred; but for some others, like assigning failures to the appropriate developers, the latter is better. Our current metric (Section 5.2.2) follows the former belief, and hence penalizes DS-PROXIMITY on gzip. The Silhouette coefficient metric can also follow the latter belief, but human beings need to specify what failures exhibit the same failure mechanism. In this study, we stick with the former belief for consistency.

Case Study 3: Flex-2.4

The third case study with flex conveys similar observations as previous two. The two seeded faults by SIR researchers are described in Figure 5.10, and the four proximity graphs are plotted in Figure 5.11.

Again, T-PROXIMITY has the worst indexing result as indicated by Figure 5.11(a), and R-PROXIMITY has the best. The dynamic slicing-based approaches, especially DS-PROXIMITY, are comparable to R-PROXIMITY. This is disclosed by both the Silhouette coefficients and the proximity graphs.
Comparison Summary

In order to draw some conclusions about the comparison between different indexing techniques, we collect the Silhouette coefficients from Figures 5.5, 5.7 and 5.11, and summarize them in Figure 5.12. The x-axis of Figure 5.12 lists the three subjects, and the y-axis is the Silhouette coefficient.

Figure 5.12 reveals the following conclusions about the indexing effectiveness of different techniques.

- **T-PROXIMITY** is apparently the weakest. This is a well-expected result because the signature function in T-PROXIMITY includes the profile of the entire execution, but the majority of the profile is essentially fault-irrelevant.

- **R-PROXIMITY** appears to be one of the best approaches. Its Silhouette coefficient is the highest in two of the three case studies, although the case with gzip is arguable. In addition, one should
note that R-PROXIMITY assumes the availability of a sufficient number of passing executions; otherwise, its indexing capability will seriously degrade [62].

- DS-PROXIMITY and FS-PROXIMITY are generally comparable to R-PROXIMITY, and neither of them requires any passing executions as R-PROXIMITY does. Sometimes, DS-PROXIMITY even outperforms R-PROXIMITY (e.g., in the case study of grep). Therefore, DS-PROXIMITY is a good alternative for R-PROXIMITY, especially when a sufficient number of passing executions are not available.

![Summary of Comparison between Different Proximities](image)

**Figure 5.12: Summarized Comparison**

### 5.4.3 Indexing Effectiveness of Pruned Slices

In this subsection, we examine whether pruned slices can help leverage the indexing quality of FS-PROXIMITY and DS-PROXIMITY. In principle, as pruned slices eliminate statements that are involved in producing correct outputs, better indexing capability should be expected. However, one needs to note that the information used in pruning slices is from the same failing execution. In comparison, R-PROXIMITY needs a separate set of passing executions to index program failures.

Proximity graphs generated by PFS-PROXIMITY and PDS-PROXIMITY on the three subject programs are presented in Figure 5.13, which suggests that pruned slices improve the indexing
quality on \texttt{gzip} and \texttt{flex}, but not on \texttt{grep}. Explicitly, on both \texttt{gzip} and \texttt{flex}, all failures form into several dense and cohesive clusters, which are visually, as well as by Silhouette coefficients, better than FS-Proximity and DS-Proximity.

We also manually check a few red crosses near blue clusters, and a few blue circles near red clusters. Our investigation shows that they are actually due to the limitation of dynamic slicing. For these failures, the first observed incorrect output is actually not wrong. The entire result is incorrect because some values are missing due to the faults. Therefore, the computed slice is unfortunately a slice on a \textit{correct} value. How to overcome cases like this appears to be a very interesting topic to explore in the future.

5.5 Discussion

In this section, we review related work, and discuss potential threats to validity of the experiment.

5.5.1 Related Work

Failing indexing, although not yet formally studied, has been a widely supported functionality in bug tracking systems [70]. A bug tracking system supports bug diagnosis and software evolution by keeping records of reported failures. Some bug tracking systems, like Bugzilla [1], are designed for manual failure reporting. Software developers or technically savvy people manually type in critical attributes of encountered failures. Typical attributes include, but are not limited to, the platform, failure stack trace, and the submitter-perceived severity. By storing the reported information into databases, failure indexing on the provided attributes is automatically supported. For example, one can easily retrieve all failures that manifest on FreeBSD and have a severity level of 5. However, such borrowed indexing capability from databases does not support automated failure prioritization and duplicate removal because root causes are usually not reported, and automatically inferring the root cause from the reported static failure data is extremely hard. In comparison, this paper, as well as previous studies [78, 63], investigates how to index program failure by program dynamic data.

On the other hand, some bug tracking systems aim at automated collection of program failures from production runs [2, 4, 60], which save users’ hassles in providing technical details. Given that
current systems have done a great job in indexing crashing failures, this paper investigates how to index noncrashing failures that will prevail in the future.

In this paper, we compare our dynamic slicing-based approach to existing techniques T-Proximity [78] and R-Proximity [63]. T-Proximity is inspired by the preceding studies that suggest program failures can be found from a set of mostly passing executions through clustering execution profiles [30, 31]. In comparison, our approach indexes program failures through dynamic slices, which are more fault-relevant than the execution profile used by T-Proximity. In comparison with R-Proximity, our approach eliminates the need of passing executions, and is shown to achieve comparable result as R-Proximity. Interestingly, similar to R-Proximity, the dynamic slicing-based approach also falls into the fault localization-based framework [63], because dynamic slicing is also a fault localization technique. Our approach is better than R-Proximity because dynamic slicing does not need any passing executions while the Sober [62] algorithm leveraged by R-Proximity does.

Recently, the importance of failure indexing is also recognized by computer system researchers [25, 92, 91]. Cohen et al. suggest that as computer systems become increasingly complex, indices of system states are helpful for system maintenance and malfunction diagnosis [25]. Basically, system statistics, such as the average CPU and memory usage, is treated as the signature of system states during a time interval. If a state is known faulty or will eventually lead to a faulty state, it is put into the index together with patches. In the future, when a similar state is encountered, corresponding patches can be automatically retrieved from the index. This approach is shown particularly effective in diagnosing performance problems [24], which are essentially noncrashing failures. Similar work is also seen on Windows platform, where snapshots of Windows registry are treated as signatures of system states. Some tools, such as STRIDER [92] and PeerPressure [91] have been invented, which leverage the signature indices to troubleshoot misconfigurations, which are another form of noncrashing failures. In comparison, our dynamic slicing-based approach focuses on indexing program failures, rather than indexing failures in a computer system, but the dynamic slicing idea can be extended to indexing system problems because intensive dependences are also involved in system problems [24].

Finally, this study also relates to dynamic program slicing. Dynamic slicing [51, 6] is a debugging
technique that captures the executed statements that are involved in computation of a wrong value. Dynamic dicing [66] leverages multiple dynamic slices to reduce the fault candidate set. The idea of dynamic dicing is to take away the statements that appear in the dynamic slices of correct values from a dynamic slice of some incorrect value. Confidence analysis [97] is a technique that estimates the likelihood of a statement being faulty by looking at its appearances in correct and incorrect dynamic slices within one execution. The goal of these techniques is to locate the root cause of a failure more precisely. Therefore, data slices may not be a good starting point for dicing and confidence analysis because they often miss the root cause. In contrast, the proposed technique uses multiple dynamic slices for the purpose of failure indexing, where the capability of discriminate failures from different groups is more important than the fault localization effectiveness. Finally, to the best of our knowledge, this is the first attempt to study the effectiveness of various types of dynamic slices in failure indexing.

5.5.2 Threats to Validity

A number of threats to validity need to be considered for the experiment results. First, although the six faults in grep, gzip and flex mimic realistic “off-by-one” and “subclause-missing” errors, they are nevertheless seeded by either ourselves or the SIR researchers [32]. For this reason, case studies with real-world faults are needed in the future. However, as this paper aims at a comparative study between different indexing techniques, seeded faults likely suffice. Especially, the two seeded faults in gzip are very complicated as shown in Section 5.4.2. Second, hand-crafted test inputs, rather than operational traces from the wild, are used in this study. In general, traces from the wild could be more complicated. But as dynamic slicing has been shown effective in extracting fault-relevant information from long executions [99], we expect similar observations about failure indexing will be made. Finally, the experiment in this paper is evaluated with the metric proposed in Section 5.2.2. Although every effort has been exercised to keep it objective and reasonable, the metric is by no means the ultimate measure. Ultimately, all indexing techniques need to be subjected to real-world noncrashing failures, and let the end-users, i.e., the developers, to judge the effectiveness.
5.6 Conclusion

In this study, we proposed a dynamic slicing-based approach to indexing noncrashing failures, an increasingly critical problem due to the dominance of semantic bugs in the future. We performed three case studies with *grep*, *gzip*, and *flex*, and the result clearly indicated the advantages of our proposed approach. Specifically, our proposed approach is more effective than T-PROXIMITY, and does not rely on correct as R-PROXIMITY does. During this study, a few interesting observations have been made, which merit further study in the future.
Figure 5.13: Effectiveness of Pruned Slices
Chapter 6

Conclusion

In this thesis, we presented three dynamic analysis techniques to automate software debugging and program failure triage, two of the most important problems in improving software quality. In particular, the proposed statistical debugging algorithm SOBER can automatically localize software faults without any prior knowledge, and as evaluated on a debugging benchmark and four median-sized real programs, SOBER is shown as one of the most accurate debugging algorithm so far. Secondly, we described a novel approach to program failure triage, which for the first time leverages existing debugging algorithms for failure triage. Previously, all dynamic analysis has been performed in the program behavior space, and the proposed approach discovers that by using automated debugging algorithms some program analysis tasks, like failure triage, can be performed in the likely-bug-location space. Recently, we have seen an interesting work which implements the same idea for parallel debugging [45]. Finally, we present another approach to failure indexing, which is based on program dynamic slicing. To some extent, the dynamic slicing-based approach completes the study of failure indexing in the sense that dynamic slices close the gap between the behavior space and the likely-bug-location space because dynamic slices can be taken as either program behaviors (all executed statements) or likely bug locations if processed accordingly. Our study of using dynamic slices as failure signature not only complements existing study on failure triage, but demonstrates how statistical analysis of multiple slices can contribute to the improvement of software quality.

While this thesis has been focused on improving software quality, I have explored some other interesting topics during my Ph.D. study. For example, I developed a software plagiarism detection tool, called GPlag, which is scalable in detecting core-part plagiarism in software [61]. Core-part plagiarism refers to the situation that only a small percentage of code is illegally re-used from program A while writing program B. For example, some person or a less principled corporation may plagiarize from certain open-source projects. Our tool can be scalable because we developed
a statistical filter to exclude code that is unlikely plagiarism and leave a small portion of code for
detailed plagiarism analysis. Also I collaborated with information retrieval (IR) researchers, and
co-devised the first probabilistic model to extract spatiotemporal theme patterns from weblog data [71]. Some interesting patterns discovered by our algorithms includes how residence in different states react to Hurricane Katrina at different time. Unfortunately, since these topics do not align well with the theme of this thesis, they are not included here. Interested readers are more than welcome to check them out.
References


Author’s Biography

Chao Liu was born in Jan., 1981 in Kaifeng, China. He was admitted by Peking University in 1999, and obtained his Bachelor of Science degree in Computer Science from Peking University in 2003. He attended University of Illinois at Urbana-Champaign in Fall 2003 to pursue Ph.D. degree in the Department of Computer Science. He obtained the Master of Science in Computer Science in 2005 on his way to Ph.D., and his Master Thesis won the David Kuck Outstanding M.S. Thesis Award.